Singular

A Computer Algebra System for Polynomial Computations

Manual

Version 4-1-1, 2018

SINGULAR is created and its development is directed and coordinated by
W. Decker, G.-M. Greuel, G. Pfister, and H. Schönemann

Principal developers:
O. Bachmann, M. Brickenstein, W. Decker, A. Frühbis-Krüger, K. Krüger,
V. Levandovskyy, C. Lossen, W. Neumann, W. Pohl, J. Schmidt, M. Schulze,
T. Siebert, R. Stobbe, E. Westenberger, T. Wichmann, O. Wienand

Fachbereich Mathematik
Zentrum für Computeralgebra
Universität Kaiserslautern
D-67653 Kaiserslautern

Short Contents

1 Preface

Please send any comments or bug reports to `singular@mathematik.uni-kl.de`.

If you want to be informed of new releases, please register as a Singular user by sending an email to `singular@mathematik.uni-kl.de` with subject line `register` and body containing the following data: your name, email address, organisation, country and platform(s).

For information on how to cite Singular see
`http://www.singular.uni-kl.de/index.php/how-to-cite-singular`.

You can also support Singular by informing us about your result obtained by using Singular.

Availability

The latest information regarding the status of Singular is always available from `http://www.singular.uni-kl.de`. The program Singular and the above mentioned parts are available via anonymous ftp through the following addresses:

GMP, libreadline
: © Free Software Foundation
`http://gmplib.org`

NTL
: © Victor Shoup
`http://www.shoup.net/ntl`

cdd (C implementation of the Double Description Method of Motzkin et al)
© Komei Fukuda
`http://www.ifor.math.ethz.ch/~fukuda/cdd_home`

FLINT © Bill Hart, Sebastian Pancratz, Fredrik Johansson
`http://www.flintlib.org`

gfanlib © Anders Jensen
`http://home.imf.au.dk/jensen/software/gfan/gfan.html`

Singular-Factory
© Gert-Martin Greuel/Rüdiger Stobbe/Martin Lee, University of Kaiserslautern:
`ftp://www.mathematik.uni-kl.de/pub/Math/Singular/Factory`

Singular-libfac
© Messollen, University of Saarbrücken:
`ftp://www.mathematik.uni-kl.de/pub/Math/Singular/Libfac`

Singular binaries and sources
`ftp://www.mathematik.uni-kl.de/pub/Math/Singular/` or via a WWW browser from `http://www.mathematik.uni-kl.de/ftp/pub/Math/Singular/`

Cygwin `http://www.cygwin.com/`

Xemacs `http://www.xemacs.org`

Some external programs are optional:

4ti2 (used by sing4ti2.lib, see Section D.4.27 [sing4ti2_lib], page 760)
`http://www.4ti2.de/`

gfan (used by tropical.lib, see Section D.13.4 [tropical_lib], page 817)
`http://home.imf.au.dk/jensen/software/gfan/gfan.html`

graphviz (used by resgraph.lib, see Section D.5.6 [resgraph_lib], page 768)
`http://www.graphviz.org/`

normaliz (used by normaliz.lib, see Section D.4.19 [normaliz_lib], page 753)
© Winfried Bruns and Bogdan Ichim
`http://www.mathematik.uni-osnabrueck.de/normaliz/`

polymake (used by polymake.lib, see Section D.13.2 [polymake_lib], page 814)
© Ewgenij Gawrilow and Michael Joswig
`http://polymake.org/`

surf (used by surf.lib, see Section D.9.3 [surf_lib], page 795)
© Stephan Endrass
`http://surf.sf.net`

surfer (used by surf.lib, see Section D.9.3 [surf_lib], page 795)
`http://www.imaginary2008.de/surfer.php`

surfex (used by surfex.lib, see Section D.9.4 [surfex_lib], page 795)
© Oliver Labs (2001-2008), Stephan Holzer (2004-2005)
`https://github.com/Singular/Sources/tree/spielwiese/Singular/LIB/surfex`

TOPCOM (used by polymake.lib, see Section D.13.2 [polymake_lib], page 814)
© Jörg Rambau
`http://www.rambau.wm.uni-bayreuth.de/TOPCOM/`

Acknowledgements

The development of Singular is directed and coordinated by Wolfram Decker, Gert-Martin Greuel, Gerhard Pfister, and Hans Schönemann.

Current devteams: Abdus Salam School of Mathematical Sciences in Lahore, BTU Cottbus, Center for Advanced Security Research Darmstadt (CASED), FU Berlin, Isfahan University of Technology, Mathematisches Forschunginstitut Oberwolfach, Oklahoma State University, RWTH Aachen, Universidad de Buenos Aires, Université de Versailles Saint-Quentin-en-Yvelines, University of Göttingen, University of Hannover, University of La Laguna and University of Valladolid.

Current Singular developers: Wolfram Decker, Gert-Martin Greuel, Gerhard Pfister, Hans Schönemann,

Shawki Al-Rashed, Daniel Andres, Mohamed Barakat, Isabel Bermejo, Muhammad Asan Binyamin, René Birkner, Rocio Blanco, Xenia Bogomolec, Michael Brickenstein, Stanislav Bulygin, Antonio Campillo, Raza Choudery, Alexander Dreyer, Christian Eder, Santiago Encinas, Jose Ignacio Farran, Anne Frühbis-Krüger, Rosa de Frutos, Eva Garcia-Llorente, Ignacio Garcia-Marco, Christian Haase, Amir Hashemi, Fernando Hernando, Bradford Hovinen, Nazeran Idress, Anders Jensen, Lars Kastner, Junaid Alan Khan, Kai Krüger, Santiago Laplagne, Grégoire Lecerf, Martin Lee, Viktor Levandovskyy, Benjamin Lorenz, Christoph Lossen, Thomas Markwig, Hannah Markwig, Irene Marquez, Bernd Martin, Edgar Martinez, Martin Monerjan, Francisco Monserrat, Oleksandr Motsak, Andreas Paffenholz, Maria Jesus Pisabarro, Diego Ruano, Afshan Sadiq, Kristina Schindelar, Mathias Schulze, Frank Seelisch, Andreas Steenpaß, Stefan Steidel, Grischa Studzinski, Katharina Werner and Eva Zerz.

Further contributions to Singular have been made by: Martin Albrecht, Olaf Bachmann, Muhammad Ahsan Banyamin, Thomas Bauer, Thomas Bayer, Markus Becker, J. Boehm, Gergo Gyula Borus, Winfried Bruns, Fernando Hernando Carrillo, Victor Castellanos, Nadine Cremer, Michael Cuntz, Kai Dehmann, Christian Dingler, Marcin Dumnicki, Stephan Endraß, Vladimir Gerdt, Philippe Gimenez, Christian Gorzel, Hubert Grassmann, Jan Hackfeld, Agnes Heydtmann, Dietmar Hillebrand, Tobias Hirsch, Markus Hochstetter, N. Idrees, Manuel Kauers, Simon King, Sebastian Jambor, Oliver Labs, Anen Lakhal, Martin Lamm, Francisco Javier Lobillo, Christoph Mang, Michael Meßollen, Andrea Mindnich, Antonio Montes, Jorge Martin Morales, Thomas Nüßler, Wolfgang Neumann, Markus Perling, Wilfried Pohl, Adrian Popescu, Tetyana Povalyaeva, Carlos Rabelo, Philipp Renner, J.-J.Salazar-Gonzalez, Alfredo Sanchez-Navarro, Ivor Saynisch, Jens Schmidt, Thomas Siebert, Christof Soeger, Silke Spang, William Stein, Rüdiger Stobbe, Henrik Strohmayer, Christian Stussak, Imade Sulandra, Akira Suzuki, Christine Theis, Enrique Tobis, Alberto Vigneron-Tenorio, Moritz Wenk, Eric Westenberger, Tim Wichmann, Oliver Wienand, Denis Yanovich and Oleksandr Yena.

We should like to acknowledge the financial support given by the Volkswagen-Stiftung, the Deutsche Forschungsgemeinschaft and the Stiftung für Innovation des Landes Rheinland-Pfalz to the Singular project.

Singular is supported by Project B5 of SFB-TRR 195 'Symbolic Tools in Mathematics and their Application'.

2 Introduction

2.1 Background

Singular is a Computer Algebra system for polynomial computations with emphasis on the special needs of commutative algebra, algebraic geometry, and singularity theory.

Singular's main computational objects are ideals and modules over a large variety of baserings. The baserings are polynomial rings or localizations thereof over a field (e.g., finite fields, the rationals, floats, algebraic extensions, transcendental extensions) or over a limited set of rings, or over quotient rings with respect to an ideal.

Singular features one of the fastest and most general implementations of various algorithms for computing Groebner resp. standard bases. The implementation includes Buchberger's algorithm (if the ordering is a wellordering) and Mora's algorithm (if the ordering is a tangent cone ordering) as special cases. Furthermore, it provides polynomial factorization, resultant, characteristic set and gcd computations, syzygy and free-resolution computations, and many more related functionalities.

Based on an easy-to-use interactive shell and a C-like programming language, Singular's internal functionality is augmented and user-extendible by libraries written in the Singular programming language. A general and efficient implementation of communication links allows Singular to make its functionality available to other programs.

Singular's development started in 1984 with an implementation of Mora's Tangent Cone algorithm in Modula-2 on an Atari computer (K.P. Neuendorf, G. Pfister, H. Schönemann; Humboldt-Universität zu Berlin). The need for a new system arose from the investigation of mathematical problems coming from singularity theory which none of the existing systems was able to handle.

In the early 1990s Singular's "home-town" moved to Kaiserslautern, a general standard basis algorithm was implemented in C and Singular was ported to Unix, MS-DOS, Windows NT, and MacOS.

Continuous extensions (like polynomial factorization, gcd computations, links) and refinements led in 1997 to the release of Singular version 1.0 and in 1998 to the release of version 1.2 (with a much faster standard and Groebner bases computation based on Hilbert series and on an improved implementation of the core algorithms, libraries for primary decomposition, ring normalization, etc.)

For the highlights of the new Singular version 4-1-1, see Section 8.1 [News and changes], page 869.

2.2 How to use this manual

For the impatient user

In Section 2.3 [Getting started], page 6, some simple examples explain how to use Singular in a step-by-step manner.

Appendix A [Examples], page 627 should come next for real learning-by-doing or to quickly solve some given mathematical problem without dwelling too deeply into Singular. This chapter contains a lot of real-life examples and detailed instructions and explanations on how to solve mathematical problems using Singular.

For the systematic user

In Chapter 3 [General concepts], page 15, all basic concepts which are important to use and to understand SINGULAR are developed. But even for users preferring the systematic approach it will be helpful to take a look at the examples in Section 2.3 [Getting started], page 6, every now and then. The topics in the chapter are organized more or less in the natural order in which the novice user is expected to have to deal with them.

- In Section 3.1 [Interactive use], page 15, and its subsections there are some words on entering and exiting SINGULAR, followed by a number of other aspects concerning the interactive user-interface.
- To do anything more than trivial integer computations, one needs to define a basering in SINGULAR. This is explained in detail in Section 3.3 [Rings and orderings], page 30.
- An overview of the algorithms implemented in the kernel of SINGULAR is given in Section 3.4 [Implemented algorithms], page 36.
- In Section 3.5 [The SINGULAR language], page 40, language specific concepts are introduced, such as the notions of names and objects, data types and conversion between them, etc.
- In Section 3.6 [Input and output], page 47, SINGULAR's mechanisms to store and retrieve data are discussed.
- The more complex concepts of procedures and libraries as well as tools for debugging them are considered in the following sections: Section 3.7 [Procedures], page 51, Section 3.8 [Libraries], page 55, and Section 3.9 [Debugging tools], page 68.

Chapter 4 [Data types], page 73, is a complete treatment of SINGULAR's data types in alphabetical order, where each section corresponds to one data type. For each data type, its purpose is explained, the syntax of its declaration is given, related operations and functions are listed, and one or more examples illustrate its usage.

Chapter 5 [Functions and system variables], page 154, is an alphabetically ordered reference list of all of SINGULAR's functions, control structures, and system variables. Each entry includes a description of the syntax and semantics of the item being explained as well as one or more examples on how to use it.

Miscellaneous

Chapter 6 [Tricks and pitfalls], page 304, is a loose collection of limitations and features which may be unexpected by those who expect the SINGULAR language to be an exact copy of the C programming language or of some other Computer Algebra system's language. Additionally, some mathematical hints are collected there.

Appendix C [Mathematical background], page 702, introduces some of the mathematical notions and definitions used throughout this manual. For example, if in doubt what exactly a "negative degree reverse lexicographical ordering" is in SINGULAR, one should refer to this chapter.

Appendix D [SINGULAR libraries], page 721, lists the libraries which come with SINGULAR, and all functions contained in them.

Typographical conventions

Throughout this manual, the following typographical conventions are adopted:

- text in `typewriter` denotes SINGULAR input and output as well as reserved names:

  The basering can, e.g., be set using the command `setring`.
- the arrow $\mapsto$ denotes SINGULAR output:

```
poly p=x+y+z;
p*p;
↦ x2+2xy+y2+2xz+2yz+z2
```

- square brackets are used to denote parts of syntax descriptions which are optional:

  [optional_text] required_text

- keys are denoted using typewriter, for example:

  `N` (press the key `N` to get to the next node in help mode)

  `RETURN` (press `RETURN` to finish an input line)

  `CTRL-P` (press the control key together with the key `P` to get the previous input line)

2.3 Getting started

Singular is a special purpose system for polynomial computations. Hence, most of the powerful computations in Singular require the prior definition of a ring. Most important rings are polynomial rings over a field, localizations thereof, or quotient rings of such rings modulo an ideal. However, some simple computations with integers (machine integers of limited size) and manipulations of strings can be carried out without the prior definition of a ring.

2.3.1 First steps

Once Singular is started, it awaits an input after the prompt `>`. Every statement has to be terminated by `;` .

```
37+5;
↦ 42
```

All objects have a type, e.g., integer variables are defined by the word `int`. An assignment is made using the symbol `=` .

```
int k = 2;
```

Test for equality resp. inequality is done using `==` resp. `!=` (or `<>`), where `0` represents the boolean value FALSE, and any other value represents TRUE.

```
k == 2;
↦ 1
k != 2;
↦ 0
```

The value of an object is displayed by simply typing its name.

```
k;
↦ 2
```

On the other hand, the output is suppressed if an assignment is made.

```
int j;
j = k+1;
```

The last displayed (!) result can be retrieved via the special symbol `_` .

```
2*_;   // the value from k displayed above
↦ 4
```

Text starting with `//` denotes a comment and is ignored in calculations, as seen in the previous example. Furthermore Singular maintains a history of the previous lines of input, which may be accessed by `CTRL-P` (previous) and `CTRL-N` (next) or the arrows on the keyboard.

The whole manual is available online by typing the command `help;` . Documentation on single topics, e.g., on `intmat`, which defines a matrix of integers, is obtained by

```
help intmat;
```

This will display the text of Section 4.7 [intmat], page 89, in the printed manual.

Next, we define a 3×3 matrix of integers and initialize it with some values, row by row from left to right:

```
intmat m[3][3] = 1,2,3,4,5,6,7,8,9;
m;
```

A single matrix entry may be selected and changed using square brackets `[` and `]`.

```
m[1,2]=0;
m;
↦ 1,0,3,
↦ 4,5,6,
↦ 7,8,9
```

To calculate the trace of this matrix, we use a `for` loop. The curly brackets `{` and `}` denote the beginning resp. end of a block. If you define a variable without giving an initial value, as the variable `tr` in the example below, Singular assigns a default value for the specific type. In this case, the default value for integers is `0`. Note that the integer variable `j` has already been defined above.

```
int tr;
for ( j=1; j <= 3; j++ ) { tr=tr + m[j,j]; }
tr;
↦ 15
```

Variables of type string can also be defined and used without having an active ring. Strings are delimited by `"` (double quotes). They may be used to comment the output of a computation or to give it a nice format. If a string contains valid Singular commands, it can be executed using the function `execute`. The result is the same as if the commands would have been written on the command line. This feature is especially useful to define new rings inside procedures.

```
"example for strings:";
↦ example for strings:
string s="The element of m ";
s = s + "at position [2,3] is:";  // concatenation of strings by +
s , m[2,3] , ".";
↦ The element of m at position [2,3] is: 6 .
s="m[2,1]=0; m;";
execute(s);
↦ 1,0,3,
↦ 0,5,6,
↦ 7,8,9
```

This example shows that expressions can be separated by `,` (comma) giving a list of expressions. Singular evaluates each expression in this list and prints all results separated by spaces.

2.3.2 Rings and standard bases

In order to compute with objects such as ideals, matrices, modules, and polynomial vectors, a ring has to be defined first.

```
ring r = 0,(x,y,z),dp;
```

The definition of a ring consists of three parts: the first part determines the ground field, the second part determines the names of the ring variables, and the third part determines the monomial ordering to be used. Thus, the above example declares a polynomial ring called `r` with a ground field of characteristic 0 (i.e., the rational numbers) and ring variables called `x`, `y`, and `z`. The `dp` at the end determines that the degree reverse lexicographical ordering will be used.

Other ring declarations:

`ring r1=32003,(x,y,z),dp;`
: characteristic 32003, variables `x`, `y`, and `z` and ordering `dp`.

`ring r2=32003,(a,b,c,d),lp;`
: characteristic 32003, variable names `a`, `b`, `c`, `d` and lexicographical ordering.

`ring r3=7,(x(1..10)),ds;`
: characteristic 7, variable names `x(1)`,. . .,`x(10)`, negative degree reverse lexicographical ordering (`ds`).

`ring r4=(0,a),(mu,nu),lp;`
: transcendental extension of Q by a , variable names `mu` and `nu`, lexicographical ordering.

`ring r5=real,(a,b),lp;`
: floating point numbers (single machine precision), variable names `a` and `b`.

`ring r6=(real,50),(a,b),lp;`
: floating point numbers with precision extended to 50 digits, variable names `a` and `b`.

`ring r7=(complex,50,i),(a,b),lp;`
: complex floating point numbers with precision extended to 50 digits and imaginary unit `i`, variable names `a` and `b`.

`ring r8=integer,(a,b),lp;`
: the ring of integers (see Section 3.3.4 [Coefficient rings], page 36), variable names `a` and `b`.

`ring r9=(integer, 60),(a,b),lp;`
: the ring of integers modulo 60 (see Section 3.3.4 [Coefficient rings], page 36), variable names `a` and `b`.

`ring r10=(integer, 2, 10),(a,b),lp;`
: the ring of integers modulo 2^10 (see Section 3.3.4 [Coefficient rings], page 36), variable names `a` and `b`.

Typing the name of a ring prints its definition. The example below shows that the default ring in Singular is $Z/32003[x, y, z]$ with degree reverse lexicographical ordering:

```
ring r11;
r11;
↦ // coefficients: ZZ/32003
↦ // number of vars : 3
↦ //        block   1 : ordering dp
↦ //                  : names    x y z
↦ //        block   2 : ordering C
```

Defining a ring makes this ring the current active basering, so each ring definition above switches to a new basering. The concept of rings in Singular is discussed in detail in Section 3.3 [Rings and orderings], page 30.

The basering is now `r11`. Since we want to calculate in the ring `r`, which we defined first, we need to switch back to it. This can be done using the function `setring`:

```
setring r;
```

Once a ring is active, we can define polynomials. A monomial, say x^3, may be entered in two ways: either using the power operator `^`, writing `x^3`, or in short-hand notation without operator, writing `x3`. Note that the short-hand notation is forbidden if a name of the ring variable(s) consists of more than one character(see Section 6.4 [Miscellaneous oddities], page 308 for details). Note, that SINGULAR always expands brackets and automatically sorts the terms with respect to the monomial ordering of the basering.

```
poly f =  x3+y3+(x-y)*x2y2+z2;
f;
↦ x3y2-x2y3+x3+y3+z2
```

The command `size` retrieves in general the number of entries in an object. In particular, for polynomials, `size` returns the number of monomials.

```
size(f);
↦ 5
```

A natural question is to ask if a point, e.g., `(x,y,z)=(1,2,0)`, lies on the variety defined by the polynomials `f` and `g`. For this we define an ideal generated by both polynomials, substitute the coordinates of the point for the ring variables, and check if the result is zero:

```
poly g =  f^2 *(2x-y);
ideal I = f,g;
ideal J = subst(I,var(1),1);
J = subst(J,var(2),2);
J = subst(J,var(3),0);
J;
↦ J[1]=5
↦ J[2]=0
```

Since the result is not zero, the point `(1,2,0)` does not lie on the variety `V(f,g)`.

Another question is to decide whether some function vanishes on a variety, or in algebraic terms, if a polynomial is contained in a given ideal. For this we calculate a standard basis using the command `groebner` and afterwards reduce the polynomial with respect to this standard basis.

```
ideal sI = groebner(f);
reduce(g,sI);
↦ 0
```

As the result is `0` the polynomial `g` belongs to the ideal defined by `f`.

The function `groebner`, like many other functions in SINGULAR, prints a protocol during calculations, if desired. The command `option(prot);` enables protocolling whereas `option(noprot);` turns it off. Section 5.1.108 [option], page 230, explains the meaning of the different symbols printed during calculations.

The command `kbase` calculates a basis of the polynomial ring modulo an ideal, if the quotient ring is finite dimensional. As an example we calculate the Milnor number of a hypersurface singularity in the global and local case. This is the vector space dimension of the polynomial ring modulo the Jacobian ideal in the global case resp. of the power series ring modulo the Jacobian ideal in the local case. See Section A.4.2 [Critical points], page 664, for a detailed explanation.

The Jacobian ideal is obtained with the command `jacob`.

```
ideal J = jacob(f);
↦ // ** redefining J **
J;
↦ J[1]=3x2y2-2xy3+3x2
↦ J[2]=2x3y-3x2y2+3y2
↦ J[3]=2z
```

SINGULAR prints the line `// ** redefining J **`. This indicates that we had previously defined a variable with name `J` of type ideal (see above).

To obtain a representing set of the quotient vector space we first calculate a standard basis, and then apply the function `kbase` to this standard basis.

```
J = groebner(J);
ideal K = kbase(J);
K;
↦ K[1]=y4
↦ K[2]=xy3
↦ K[3]=y3
↦ K[4]=xy2
↦ K[5]=y2
↦ K[6]=x2y
↦ K[7]=xy
↦ K[8]=y
↦ K[9]=x3
↦ K[10]=x2
↦ K[11]=x
↦ K[12]=1
```

Then

```
size(K);
↦ 12
```

gives the desired vector space dimension $K[x, y, z]/\text{jacob}(f)$. As in Singular the functions may take the input directly from earlier calculations, the whole sequence of commands may be written in one single statement.

```
size(kbase(groebner(jacob(f))));
↦ 12
```

When we are not interested in a basis of the quotient vector space, but only in the resulting dimension we may even use the command `vdim` and write:

```
vdim(groebner(jacob(f)));
↦ 12
```

2.3.3 Procedures and libraries

Singular offers a comfortable programming language, with a syntax close to C. So it is possible to define procedures which bind a sequence of several commands in a new one. Procedures are defined using the keyword `proc` followed by a name and an optional parameter list with specified types. Finally, a procedure may return a value using the command `return`.

We may e.g. define the following procedure called `Milnor`: (Here the parameter list is `(poly h)` meaning that `Milnor` must be called with one argument which can be assigned to the type `poly` and is refered to by the name `h`.)

```
proc Milnor (poly h)
{
  return(vdim(groebner(jacob(h))));
}
```

Note: if you have entered the first line of the procedure and pressed `RETURN`, Singular prints the prompt `.` (dot) instead of the usual prompt `>` . This shows that the input is incomplete and Singular expects more lines. After typing the closing curly bracket, Singular prints the usual prompt indicating that the input is now complete.

Then we can call the procedure:

```
Milnor(f);
↦ 12
```

Note that the result may depend on the basering as we will see in the next chapter.

The distribution of SINGULAR contains several libraries, each of which is a collection of useful procedures based on the kernel commands, which extend the functionality of SINGULAR. The command `listvar(package);` list all currently loaded libraries. The command `LIB "all.lib";` loads all libraries.

One of these libraries is `sing.lib` which already contains a procedure called `milnor` to calculate the Milnor number not only for hypersurfaces but more generally for complete intersection singularities.

Libraries are loaded using the command `LIB`. Some additional information during the process of loading is displayed on the screen, which we omit here.

```
LIB "sing.lib";
```

As all input in SINGULAR is case sensitive, there is no conflict with the previously defined procedure `Milnor`, but the result is the same.

```
milnor(f);
↦ 12
```

The procedures in a library have a help part which is displayed by typing

```
help milnor;
```

as well as some examples, which are executed by

```
example milnor;
```

Likewise, the library itself has a help part, to show a list of all the functions available for the user which are contained in the library.

```
help sing.lib;
```

The output of the help commands is omitted here.

2.3.4 Change of rings

To calculate the local Milnor number we have to do the calculation with the same commands in a ring with local ordering. We can define the localization of the polynomial ring at the origin (see Appendix B [Polynomial data], page 695, and Appendix C [Mathematical background], page 702).

```
ring rl = 0,(x,y,z),ds;
```

The ordering directly affects the standard basis which will be calculated. Fetching the polynomial defined in the ring `r` into this new ring, helps us to avoid retyping previous input.

```
poly f = fetch(r,f);
f;
↦ z2+x3+y3+x3y2-x2y3
```

Instead of `fetch` we can use the function `imap` which is more general but less efficient. The most general way to fetch data from one ring to another is to use maps, this will be explained in Section 4.11 [map], page 104.

In this ring the terms are ordered by increasing exponents. The local Milnor number is now

```
Milnor(f);
↦ 4
```

This shows that `f` has outside the origin in affine 3-space singularities with local Milnor number adding up to $12 - 4 = 8$. Using global and local orderings as above is a convenient way to check whether a variety has singularities outside the origin.

The command `jacob` applied twice gives the Hessian of `f`, in our example a 3x3 - matrix.

```
matrix H = jacob(jacob(f));
H;
↦ H[1,1]=6x+6xy2-2y3
↦ H[1,2]=6x2y-6xy2
↦ H[1,3]=0
↦ H[2,1]=6x2y-6xy2
↦ H[2,2]=6y+2x3-6x2y
↦ H[2,3]=0
↦ H[3,1]=0
↦ H[3,2]=0
↦ H[3,3]=2
```

The `print` command displays the matrix in a nicer format.

```
print(H);
↦ 6x+6xy2-2y3,6x2y-6xy2,  0,
↦ 6x2y-6xy2,  6y+2x3-6x2y,0,
↦ 0,          0,          2
```

We may calculate the determinant and (the ideal generated by all) minors of a given size.

```
det(H);
↦ 72xy+24x4-72x3y+72xy3-24y4-48x4y2+64x3y3-48x2y4
minor(H,1);  // the 1x1 - minors
↦ _[1]=2
↦ _[2]=6y+2x3-6x2y
↦ _[3]=6x2y-6xy2
↦ _[4]=6x2y-6xy2
↦ _[5]=6x+6xy2-2y3
```

The algorithm of the standard basis computation may be affected by the command `option`. For example, a reduced standard basis of the ideal generated by the 1×1-minors of H is obtained in the following way:

```
option(redSB);
groebner(minor(H,1));
↦ _[1]=1
```

This shows that 1 is contained in the ideal of the 1×1-minors, hence the corresponding variety is empty.

2.3.5 Modules and their annihilator

Now we shall give three more advanced examples.

Singular is able to handle modules over all the rings, which can be defined as a basering. A free module of rank `n` is defined as follows:

```
ring rr;
int n = 4;
freemodule(4);
↦ _[1]=gen(1)
↦ _[2]=gen(2)
↦ _[3]=gen(3)
↦ _[4]=gen(4)
typeof(_);
↦ module
print(freemodule(4));
↦ 1,0,0,0,
↦ 0,1,0,0,
```

```
↦ 0,0,1,0,
↦ 0,0,0,1
```

To define a module, we provide a list of vectors generating a submodule of a free module. Then this set of vectors may be identified with the columns of a matrix. For that reason in Singular matrices and modules may be interchanged. However, the representation is different (modules may be considered as sparse matrices).

```
ring r =0,(x,y,z),dp;
module MD = [x,0,x],[y,z,-y],[0,z,-2y];
matrix MM = MD;
print(MM);
↦ x,y,0,
↦ 0,z,z,
↦ x,-y,-2y
```

However the submodule MD may also be considered as the module of relations of the factor module r^3/MD. In this way, Singular can treat arbitrary finitely generated modules over the basering (see Section B.1 [Representation of mathematical objects], page 695).

In order to get the module of relations of MD , we use the command `syz`.

```
syz(MD);
↦ _[1]=x*gen(3)-x*gen(2)+y*gen(1)
```

We want to calculate, as an application, the annihilator of a given module. Let $M = r^3/U$, where U is our defining module of relations for the module M.

```
module U = [z3,xy2,x3],[yz2,1,xy5z+z3],[y2z,0,x3],[xyz+x2,y2,0],[xyz,x2y,1];
```

Then, by definition, the annihilator of M is the ideal $\mathrm{ann}(M) = \{a \mid aM = 0\}$ which is, by definition of M, the same as $\{a \mid ar^3 \in U\}$. Hence we have to calculate the quotient $U{:}r^3$. The rank of the free module is determined by the choice of U and is the number of rows of the corresponding matrix. This may be retrieved by the function `nrows`. All we have to do now is the following:

```
quotient(U,freemodule(nrows(U)));
```

The result is too big to be shown here.

2.3.6 Resolution

There are several commands in Singular for computing free resolutions. The most general command is `res(... ,n)` which determines heuristically what method to use for the given problem. It computes the free resolution up to the length n , where $n = 0$ corresponds to the full resolution.

Here we use the possibility to inspect the calculation process using the option `prot`.

```
ring R;      // the default ring in char 32003
R;
↦ //   characteristic : 32003
↦ //   number of vars : 3
↦ //        block   1 : ordering dp
↦ //                  : names    x y z
↦ //        block   2 : ordering C
ideal I = x4+x3y+x2yz,x2y2+xy2z+y2z2,x2z2+2xz3,2x2z2+xyz2;
option(prot);
resolution rs = res(I,0);
↦ using lres
↦ 4(m0)4(m1).5(m1)g.g6(m1)...6(m2)..
```

Disable this protocol with

```
option(noprot);
```

When we enter the name of the calculated resolution, we get a pictorial description of the minimized resolution where the exponents denote the rank of the free modules. Note that the calculated resolution itself may not yet be minimal.

```
rs;
↦ 1      4      5      2      0
↦R  <-- R  <-- R  <-- R  <-- R
↦
↦0      1      2      3      4
print(betti(rs),"betti");
↦            0     1     2     3
↦ ------------------------------
↦     0:     1     -     -     -
↦     1:     -     -     -     -
↦     2:     -     -     -     -
↦     3:     -     4     1     -
↦     4:     -     -     1     -
↦     5:     -     -     3     2
↦ ------------------------------
↦ total:     1     4     5     2
```

In order to minimize the resolution, that is to calculate the maps of the minimal free resolution, we use the command `minres`:

```
rs=minres(rs);
```

A single module in this resolution is obtained (as usual) with the brackets `[` and `]`. The `print` command can be used to display a module in a more readable format:

```
print(rs[3]);
↦ z3,   -xyz-y2z-4xz2+16z3,
↦ 0,    -y2,
↦ -y+4z,48z,
↦ x+2z, 48z,
↦ 0,    x+y-z
```

In this case, the output is to be interpreted as follows: the 3rd syzygy module of R/I, `rs[3]`, is the rank-2-submodule of R^5 generated by the vectors $(z^3, 0, -y+4z, x+2z, 0)$ and $(-xyz-y^2z-4xz^2+16z^3, -y^2, 48z, 48z, x+y-z)$.

3 General concepts

3.1 Interactive use

In this section, aspects of interactive use are discussed. This includes how to enter and exit SINGULAR, how to interpret its prompt, how to get online help, and so on.

There are a few important notes which one should not forget:

- every command has to be terminated by a `;` (semicolon) followed by a ⟨RETURN⟩
- the online help is accessible by means of the `help` function

3.1.1 How to enter and exit

SINGULAR can either be run in an ASCII-terminal or within Emacs.

To start SINGULAR in its ASCII-terminal user interface, enter `Singular` at the system prompt. The SINGULAR banner appears which, among other data, reports the version and the compilation date.

To start SINGULAR in its Emacs user interface, either enter `ESingular` at the system prompt, or type `M-x singular` within a running Emacs (provided you have loaded the file `singular.el` in your running Emacs, see Section 3.2.2 [Running SINGULAR under Emacs], page 24 for details).

Generally, we recommend to use SINGULAR in its Emacs interface, since this offers many more features and is more convenient to use than the ASCII-terminal interface (see Section 3.2 [Emacs user interface], page 22).

To exit SINGULAR type `quit;`, `exit;` or `$` (or, when running within Emacs preferably type `C-c $`).

SINGULAR and `ESingular` may also be started with command line options and with filenames as arguments. More generally, the startup syntax is

```
Singular  [options] [file1 [file2 ...]]
ESingular  [options] [file1 [file2 ...]]
```

See Section 3.1.6 [Command line options], page 19, Section 3.1.7 [Startup sequence], page 22, Section 3.2.2 [Running SINGULAR under Emacs], page 24.

3.1.2 The SINGULAR prompt

The SINGULAR prompt `>` (larger than) asks the user for input of commands. The “continuation” prompt `.` (period) asks the user for input of missing parts of a command (e.g. the semicolon at the end of every command).

SINGULAR does not interpret the semicolon as the end of a command if it occurs inside a string. Also, SINGULAR waits for blocks (sequences of commands enclosed in curly brackets) to be closed before prompting with `>` for more commands. Thus, if SINGULAR does not respond with its regular prompt after typing a semicolon it may wait for a `"` or a `}` first.

Additional semicolons will not harm SINGULAR since they are interpreted as empty statements.

3.1.3 The online help system

The online help system is invoked by the `help` command. `?` may be used as a synonym for `help`. Simply typing `help;` displays the “top” of the help system (i.e., the title page of the SINGULAR manual) which offers a short table of contents. Typing `help` topic`;` shows the available documentation on the respective topic. Here, topic may be either a function name or, more generally, any index entry of the SINGULAR manual. Furthermore, topic may contain wildcard characters. See Section 5.1.52 [help], page 191, for more information.

Online help information can be displayed in various help browsers. The following table lists a summary of the browsers which are always present. Usually, external browsers are much more convenient: A complete, customizable list can be found in the file `LIB/help.cnf`.

Browser	**Platform**	**Description**
html	Windows	displays a html version of the manual in your default html browser
builtin	all	simply outputs the help information in plain ASCII format
emacs	Unix, Windows	when running SINGULAR within (X)emacs, displays help inside the (X)emacs info buffer.
dummy	all	displays an error message due to the non-availability of a help browser

External browsers depend on your system and the contents of `LIB/help.cnf`, the default includes:
`htmlview` (displays HTML help pages via `htlmview`),
`mac` (displays HTML help pages via `open`),
`mac-net` (displays HTML help pages via `open`),
`mozilla` (displays HTML help pages via `mozilla`),
`firefox` (displays HTML help pages via `firefox`),
`konqueror` (displays HTML help pages via `konqueror`),
`galeon` (displays HTML help pages via `galeon`),
`netscape` (displays HTML help pages via `netscape`),
`safari` (displays HTML help pages on MacOsX via `safari`),
`tkinfo` (displays INFO help pages via `tkinfo`),
`xinfo` (displays INFO help pages via `info`),
`info` (displays INFO help pages via `info`),
`lynx` (displays HTML help pages via `lynx`).

The browser which is used to display the help information, can be either set at startup time with the command line option (see Section 3.1.6 [Command line options], page 19)

```
--browser=<browser>
```

or with the SINGULAR command (see Section 5.1.151 [system], page 270)

```
system("--browser", "<browser>");
```

The SINGULAR command

```
system("browsers");
```

lists all available browsers and the command

```
system("--browser");
```

returns the currently used browser.

If no browser is explicitely set by the user, then the first available browser (w.r.t. the order of the browsers in the file `LIB/help.cnf`) is chosen.

The `.singularrc` (see Section 3.1.7 [Startup sequence], page 22) file is a good place to set your default browser. Recall that if a file `$HOME/.singularrc` exists on your system, then the content of this file is executed before the first user input. Hence, putting

```
if (! system("--emacs"))
{
  // only set help browser if not running within emacs
  system("--browser", "info");
}
// if help browser is later on set to a web browser,
// allow it to fetch HTML pages from the net
system("--allow-net", 1);
```

in your file `$HOME/.singularrc` sets your default browser to `info`, unless Singular is run within emacs (in which case the default browser is automatically set to `emacs`).

Obviously, certain external files and programs are required for the Singular help system to work correctly. If something is not available or goes wrong, here are some tips for troubleshooting the help system:

- Under Unix, the environment variable `DISPLAY` has to be set for all X11 browsers to work.
- The help browsers are only available if the respective programs are installed on your system (for `xinfo`, the programs `xterm` and `info` are necessary). You can explicitely specify which program to use, by changing the entry in `LIB/help.cnf`
- If the help browser cannot find the local html pages of the Singular manual (which it will look for at `$RootDir/html` – see Section 3.8.11 [Loading a library], page 67 for more info on `$RootDir`) *and* the (command-line) option `--allow-net` has *explicitely* been set (see Section 3.1.6 [Command line options], page 19 and Section 5.1.151 [system], page 270 for more info on setting values of command-line options), then it dispatches the html pages from `http://www.singular.uni-kl.de/Manual`. (Note that the non-local net-access of HTML pages is disabled, by default.)
  An alternative location of a local directory where the html pages reside can be specified by setting the environment variable `SINGULAR_HTML_DIR`.
- The `info` based help browsers `tkinfo`, `xinfo`, `info`, and `builtin` need the (info) file `singular.hlp` which will be looked for at `$RootDir/info/singular.hlp` (see Section 3.8.11 [Loading a library], page 67 for more info on `$RootDir`). An alternative location of the info file of the manual can be specified by setting the environment variable `SINGULAR_INFO_FILE`.

Section 3.1.6 [Command line options], page 19

Info help browsers

The help browsers `tkinfo`, `xinfo` and `info` (so-called info help browsers) are based on the `info` program from the GNU `texinfo` package. See section "Getting started" in *The Info Manual*, for more information.

For info help browsers, the online manual is decomposed into "nodes" of information, closely related to the division of the printed manual into sections and subsections. A node contains text describing a specific topic at a specific level of detail. The top line of a node is its "header". The node's header tells the name of the current node (`Node:`), the name of the next node (`Next:`), the name of the previous node (`Prev:`), and the name of the upper node (`Up:`).

To move within info, type commands consisting of single characters. Do not type `RETURN`. Do not use cursor keys, either. Using some of the cursor keys by accident might pop to some totally different node. Type `l` to return to the original node. Some of the `info` commands read input from the command line at the bottom. The `TAB` key may be used to complete partially entered input.

The most important commands are:

`q` leaves the online help system

`n` goes to the next node

`p` goes to the previous node

`u` goes to the upper node

`m` picks a menu item specified by name

`f` follows a cross reference

`l` goes to the previously visited node

`b` goes to the beginning of the current node

`e` goes to the end of the current node

`SPACE` scrolls forward a page

`DEL` scrolls backward a page

`h` invokes info tutorial (use `l` to return to the manual or `CTRL-X 0` to remove extra window)

`CTRL-H` shows a short overview over the online help system (use `l` to return to the manual or `CTRL-X 0` to remove extra window)

`s` searches through the manual for a specific string, and selects the node in which the next occurrence is found

`1`, ..., `9` picks i-th subtopic from a menu

3.1.4 Interrupting SINGULAR

On Unix-like operating systems and on Windows NT, typing `CTRL-C` (or, alternatively `C-c C-c`, when running within Emacs), interrupts SINGULAR. SINGULAR prints the current command and the current line and prompts for further action. The following choices are available:

`a` returns to the top level after finishing the current (kernel) command. Notice that commands of the SINGULAR kernel (like `std`) cannot be aborted, i.e. (`a`)bort only happens whenever the interpreter is active.

`c` continues

`q` quits SINGULAR

3.1.5 Editing input

The following keys can be used for editing the input and retrieving previous input lines:

`TAB` provides command line completion for function names and file names

`CTRL-B` moves cursor to the left

`CTRL-F` moves cursor to the right

`CTRL-A` moves cursor to the beginning of the line

`CTRL-E` moves cursor to the end of the line

`CTRL-D` deletes the character under the cursor
Warning: on an empty line, `CTRL-D` is interpreted as the `EOF` character which immediately terminates SINGULAR.

`BACKSPACE`
`DELETE`
`CTRL-H` deletes the character before the cursor

`CTRL-K` kills from cursor to the end of the line

`CTRL-U` kills from cursor to the beginning of the line

`CTRL-N` saves the current line to history and gives the next line

`CTRL-P` saves the current line to history and gives the previous line

`RETURN` saves the current line to the history and sends it to the SINGULAR parser for interpretation

When run under a Unix-like operating system and in its ASCII-terminal user interface, SINGULAR tries to dynamically link at runtime with the GNU Readline library. See section "Command Line Editing" in *The GNU Readline Library Manual*, for more information. If a shared version of this library can be found on your machine, then additional command-line editing features like history completion are available. In particuliar, if SINGULAR is able to load that library and if the environment variable `SINGULARHIST` is set and has a name of a valid file as value, then the input history is stored across sessions using this file. Otherwise, i.e., if the environment variable `SINGULARHIST` is not set, then the history of the last inputs is only available for previous commands of the current session.

3.1.6 Command line options

The startup syntax is

```
Singular  [options] [file1 [file2 ...]]
ESingular  [options] [file1 [file2 ...]]
```

Options can be given in both their long and short format. The following options control the general behaviour of SINGULAR:

`-d`, `--sdb` Enable the use of the source code debugger. See Section 3.9.3 [Source code debugger], page 69.

`-e`, `--echo[=VAL]`
Set value of variable `echo` to `VAL` (integer in the range 0, . . . , 9). Without an argument, `echo` is set to 1, which echoes all input coming from a file. By default, the value of `echo` is 0. See Section 5.3.2 [echo], page 297.

`-h`, `--help`
Print a one-line description of each command line option and exit.

`--allow-net`
Allow the help browsers based on a web browser to fetch HTML manual pages over the net from the WWW home-site of SINGULAR. See Section 3.1.3 [The online help system], page 15, for more info.

`--browser="VAL"`
Use `VAL` as browser for the SINGULAR online manual.
`VAL` may be one of the browsers mentioned in `LIB/help.cnf`, for example `html` (Windows only), `mozilla`, `firefox`, `konqueror`, `galeon`, `netscape`, `safari` (OsX only), `xinfo`, `tkinfo`, `info`, `builtin`, or `emacs`. Depending on your platform and local installation, only some browsers might be available. The default browser is `html` for Windows and one based on a web browser for Unix platforms. See Section 3.1.3 [The online help system], page 15, for more info.

`--no-rc` Do not execute the `.singularrc` file on start-up. By default, this file is executed on start-up. See Section 3.1.7 [Startup sequence], page 22.

`--no-stdlib`
: Do not load the library `standard.lib` on start-up. By default, this library is loaded on start-up. See Section 3.1.7 [Startup sequence], page 22.

`--no-warn`
: Do not display warning messages.

`--no-out` Suppress display of all output.

`--no-shell`
: Runs Singular in restricted mode to disallow shell escape commands. Objects of type link will also be unable to use.

`-t`, `--no-tty`
: Do not redefine the characteristics of the terminal. This option should be used for batch processes.

`-q`, `--quiet`
: Do not print the start-up banner and messages when loading libraries. Furthermore, redirect `stderr` (all error messages) to `stdout` (normal output channel). This option should be used if SINGULAR's output is redirected to a file.

`-v` Print extended information about the version and configuration of SINGULAR (used optional parts, compilation date, start of random generator etc.). This information should be included if a user reports an error to the authors.

The following command line options allow manipulations of the timer and the pseudo random generator and enable the passing of commands and strings to SINGULAR:

`-c`, `--execute=STRING`
: Execute `STRING` as (a sequence of) SINGULAR commands on start-up after the `.singularrc` file is executed, but prior to executing the files given on the command line. E.g., `Singular -c "help all.lib; quit;"` shows the help for the library `all.lib` and exits.

`-u`, `--user-option=STRING`
: Returns `STRING` on `system("--user-option")`. This is useful for passing arbitrary arguments from the command line to the SINGULAR interpreter. E.g., `Singular -u "xxx.dump" -c 'getdump(system("--user-option"))'` reads the file `xxx.dump` at start-up and allows the user to start working with all the objects defined in a previous session.

`-r`, `--random=SEED`
: Seed (i.e., set the initial value of) the pseudo random generator with integer `SEED`. If this option is not given, then the random generator is seeded with a time-based `SEED` (the number of seconds since January, 1, 1970, on Unix-like operating systems, to be precise).

`--min-time=SECS`
: If the `timer` (see Section 5.3.8 [timer], page 300), resp. `rtimer` (see Section 5.3.10 [rtimer], page 302) , variable is set, report only times larger than `SECS` seconds (`SECS` needs to be a floating point number greater than 0). By default, this value is set to 0.5 (i.e., half a second). E.g., the option `--min-time=0.01` forces SINGULAR to report all times larger than 1/100 of a second.

`--ticks-per-sec=TICKS`
: Set unit of timer to `TICKS` ticks per second (i.e., the value reported by the `timer` and `rtimer` variable divided by `TICKS` gives the time in seconds). By default, this value is 1.

`--cpus=CPUs`
: set the maximal number of CPUs to use.

`--cntrlc=C`
: set the default answer for interupt signals to C which should be a for abort, c for continue or q for quit.

The next three options are of interest for the use with ssi links:

`-b`, `--batch`
: Run in batch mode. Opens a TCP/IP connection with host specified by `--MPhost` at the port specified by `--MPport`. Input is read from and output is written to this connection in the format given by `--link`. See Section 4.9.5 [Ssi links], page 97.

`--MPport=PORT`
: Use `PORT` as default port number for connections (whenever not further specified). This option is mandatory when the `--batch` option is given. See Section 4.9.5 [Ssi links], page 97.

`--MPhost=HOST`
: Use `HOST` as default host for connections (whenever not further specified). This option is mandatory when the `--batch` option is given. See Section 4.9.5 [Ssi links], page 97.

Finally, the following options are only available when running `ESingular` (see Section 3.2.2 [Running SINGULAR under Emacs], page 24 for details).

`--emacs=EMACS`
: Use `EMACS` as Emacs program to run the SINGULAR Emacs interface, where `EMACS` may e.g. be emacs or xemacs.

`--emacs-dir=DIR`
: Set the singular-emacs-home-directory, which is the directory where singular.el can be found, to `DIR`.

`--emacs-load=FILE`
: Load `FILE` on Emacs start-up, instead of the default load file.

`--singular=PROG`
: Start `PROG` as SINGULAR program within Emacs

The value of options given to SINGULAR (resp. their default values, if an option was not given), can be checked with the command `system("--long_option_name")`. See Section 5.1.151 [system], page 270.

```
  system("--quiet");    // if ‘‘quiet’’ 1, otherwise 0
↦ 1
  system("--min-time"); // minimal reported time
↦ 0.5
  system("--random");   // seed of the random generator
↦ 12345678
```

Furthermore, the value of options (e.g., `--browser`) can be re-defined while SINGULAR is running using the command `system("--long_option_name_string ",expression)`. See Section 5.1.151 [system], page 270.

```
system("--browser", "builtin");  // sets browser to 'builtin'
system("--ticks-per-sec", 100);  // sets timer resolution to 100
```

3.1.7 Startup sequence

On start-up, Singular

1. loads the library `standard.lib` (provided the `--no-stdlib` option was not given),
2. searches the current directory and then the home directory of the user, and then all directories contained in the library `SearchPath` (see Section 3.8.11 [Loading a library], page 67 for more info on `SearchPath`) for a file named `.singularrc` and executes it, if found (provided the `--no-rc` option was not given),
3. executes the string specified with the `--execute` command line option,
4. executes the files `file1`, `file2` ... (given on the command line) in that order.

Note: `.singularrc` file(s) are an approriate place for setting some default values of (command-line) options.

For example, a system administrator might remove the locally installed HTML version of the manual and put a `.singularrc` file with the following content

```
if (system("version") >= 1306) // assure backwards-compatibility
{
  system("--allow-net", 1);
}; // the last semicolon is important: otherwise no ">", but "." prompt
```

in the directory containing the Singular libraries, thereby allowing to fetch the HTML on-line help from the WWW home-site of Singular.

On the other hand, a single user might put a `.singularrc` with the following content

```
if (system("version") >= 1306) // assure backwards-compatibility
{
  if (! system("--emacs"))
  {
    // set default browser to info, unless we run within emacs
    system("--browser", "info");
  }
}; // the last semicolon is important: otherwise no ">", but "." prompt
```

in his home directory, which sets the default help browser to `info` (unless Singular is run within emacs) and thereby prevents the execution of the"global" `.singularrc` file installed by the system administrator (since the `.singularrc` file of the user is found before the "global" `.singularrc` file installed by the system administrator).

3.2 Emacs user interface

Besides running Singular in an ASCII-terminal, Singular might also be run within Emacs. Emacs (or, XEmacs which is very similar) is a powerful and freely available text editor, which, among others, provides a framework for the implementation of interactive user interfaces. Starting from version 1.3.6, Singular provides such an implementation, the so-called Singular Emacs mode, or Emacs user interface.

Generally, we recommend to use the Emacs interface, instead of the ASCII-terminal interface: The Emacs interface does not only provide everything the ASCII-terminal interface provides, but offers much more. Among others, it offers

- color highlighting

- truncation of long lines
- folding of input and output
- TAB-completion for help topics
- highlighting of matching parentheses
- key-bindings and interactive menus for most user interface commands and for basic SINGULAR commands (such as loading of libraries and files)
- a mode for running interactive SINGULAR demonstrations
- convenient ways to edit SINGULAR input files
- interactive customization of nearly all aspects of the user-interface.

In order to use the SINGULAR-Emacs interface you need to have Emacs version 20 or higher, or XEmacs version 20.3 or higher installed on your system. These editors can be downloaded for most hard- and software platforms, sources from either `http://www.gnu.org/software/emacs/emacs.html` (Emacs), or from `http://www.xemacs.org` (XEmacs). (Download of binaries depend on your OS). The differences between Emacs and XEmacs w.r.t. the SINGULAR-Emacs interface are marginal – which editor to use is mainly a matter of personal preferences.

The simplest way to start-up SINGULAR in its Emacs interface is by running the program `ESingular` which is contained in the Singular distribution. Alternatively, SINGULAR can be started within an already running Emacs – see Section 3.2.2 [Running SINGULAR under Emacs], page 24 for details.

The next section gives a tutorial-like introduction to Emacs. This introductory section is followed by sections which explain the functionality of various aspects of the Emacs user interface in more detail: how to start/restart/kill SINGULAR within Emacs, how to run an interactive demonstration, how to customize the Emacs user interface, etc. Finally, the 20 most important commands of the Emacs interface together with their key bindings are listed.

3.2.1 A quick guide to Emacs

This section gives a tutorial-like introduction to Emacs. Especially to users who are not familiar with Emacs, we recommend that they go through this section and try out the described features.

Emacs commands generally involve the `CONTROL` key (sometimes labeled `CTRL` or `CTL`) or the `META` key. On some keyboards, the `META` key is labeled `ALT` or `EDIT` or something else (for example, on Sun keyboards, the diamond key to the left of the space-bar is `META`). If there is no `META` key, the `ESC` key can be used, instead. Rather than writing out `META` or `CONTROL` each time we want to prefix a character, we will use the following abbreviations:

`C-<chr>` means hold the ⟨CONTROL⟩ key while typing the character `<chr>`. Thus, `C-f` would be: hold the ⟨CONTROL⟩ key and type `f`.

`M-<chr>` means hold the ⟨META⟩ key down while typing `<chr>`. If there is no ⟨META⟩ key, type ⟨ESC⟩, release it, then type the character `<chr>`.

For users new to Emacs, we highly recommend that they go through the interactive Emacs tutorial: type `C-h t` to start it.

For others, it is important to understand the following Emacs concepts:

window In Emacs terminology, a window refers to separate panes within the same window of the window system, and not to overlapping, separate windows. When using SINGULAR within Emacs, extra windows may appear which display help or output from certain commands. The most important window commands are:

`C-x 1`	`File->Un-Split`	Un-Split window (i.e., kill other windows)
`C-x o`		Goto other window, i.e. move cursor into other window.

cursor and point
: The location of the cursor in the text is also called "point". To paraphrase, the cursor shows on the screen where point is located in the text. Here is a summary of simple cursor-moving operations:

C-f	Move forward a character
C-b	Move backward a character
M-f	Move forward a word
M-b	Move backward a word
C-a	Move to the beginning of line
C-e	Move to the end of line

buffer
: Any text you see in an Emacs window is always part of some buffer. For example, each file you are editing with Emacs is stored inside a buffer, but also SINGULAR is running inside an Emacs buffer. Each buffer has a name: for example, the buffer of a file you edit usually has the same name as the file, SINGULAR is running in a buffer which has the name *singular* (or, *singular<2>*, *singular<3>*, etc., if you have multiple SINGULAR sessions within the same Emacs).

When you are asked for input to an Emacs command, the cursor moves to the bottom line of Emacs, i.e., to a special buffer, called the "minibuffer". Typing ⟨RETURN⟩ within the minibuffer, ends the input, typing ⟨SPACE⟩ within the minibuffer, lists all possible input values to the interactive Emacs command.

The most important buffer commands are

C-x b	Switch buffer
C-x k	Kill current buffer

Alternatively, you can switch to or kill buffers using the Buffer menu.

Executing commands
: Emacs commands are executed by typing M-x <command-name> (remember that ⟨SPACE⟩ completes partial command names). Important and frequently used commands have short-cuts for their execution: Key bindings or even menu entries. For example, a file can be loaded with M-x load-file, or C-x C-f, or with the File->Open menu.

How to exit
: To end the Emacs (and, SINGULAR) session, type C-x C-c (two characters), or use the File -> Exit menu.

When Emacs hangs
: If Emacs stops responding to your commands, you can stop it safely by typing C-g, or, if this fails, by typing C-].

More help
: Nearly all aspects of Emacs are very well documented: type C-h and then a character saying what kind of help you want. For example, typing C-h i enters the Info documentation browser.

Using the mouse
: Emacs is fully integrated with the mouse. In particular, clicking the right mouse button brings up a pop-up menu which usually contains a few commonly used commands.

3.2.2 Running SINGULAR under Emacs

There are two ways to start the SINGULAR Emacs interface: Typing ESingular instead of Singular on the command shell launches a new Emacs process, initializes the interface and runs SINGULAR within Emacs. The other way is to start the interface in an already running Emacs, by typing

M-x singular inside Emacs. This initializes the interface and runs Singular within Emacs. Both ways are described in more detail below.

Note: To properly run the Emacs interface, several files are needed which usually reside in the `emacs` subdirectory of your Singular distribution. This directory is called singular-emacs-home-directory in the following.

Starting the interface using ESingular

As mentioned above, `ESingular` is an "out-of-the-box" solution: You don't have to add special things to your `.emacs` startup file to initialize the interface; everything is done for you in a special file called `.emacs-singular` (which comes along with the Singular distribution and resides in the singular-emacs-home-directory) which is automatically loaded on Emacs startup (and the loading of the `.emacs` file is automatically suppressed).

The customizable variables of the Singular Emacs interface are set to defaults which give the novice user a very shell like feeling of the interface. Nevertheless, these default settings can be changed, see Section 3.2.4 [Customization of the Emacs interface], page 27. Besides other Emacs initializations, such as fontification or blinking parentheses, a new menu item called `Singular` is added to the main menu, providing menu items for starting Singular. On XEmacs, a button starting Singular is added to the main toolbar.

The Singular interface is started automatically; once you see a buffer called `*singular*` and the Singular prompt, you are ready to start your Singular session.

`ESingular` inherits all `Singular` options. For a description of all these options, see Section 3.1.6 [Command line options], page 19. Additionally there are the following options which are special to `ESingular`:

command-line option / environment variable	functionality
`--emacs=EMACS` `ESINGULAR_EMACS`	Use `EMACS` as Emacs program to run the Singular Emacs interface, where `EMACS` may e.g. be emacs or xemacs.
`--emacs-dir=DIR` `ESINGULAR_EMACS_DIR`	Set the singular-emacs-home-directory, which is the directory where singular.el can be found, to `DIR`.
`--emacs-load=FILE` `ESINGULAR_EMACS_LOAD`	Load `FILE` on Emacs start-up, instead of the default load file.
`--singular=PROG` `ESINGULAR_SINGULAR`	Start `PROG` as Singular program within Emacs

Notice that values of these options can also be given by setting the above mentioned environment variables (where values given as command-line arguments take priority over values given by environment variables).

Starting the interface within a running Emacs

If you are a more experienced Emacs user and you already have your own local `.emacs` startup file, you might want to start the interface out of your running Emacs without using `ESingular`. For this, you should add the following lisp code to your `.emacs` file:

```
(setq load-path (cons "<singular-emacs-home-directory>" load-path))
(autoload 'singular "singular"
  "Start Singular using default values." t)
(autoload 'singular-other "singular"
  "Ask for arguments and start Singular." t)
```

Then typing `M-x singular` in a running Emacs session initializes the interface in a new buffer and launches a SINGULAR process. The SINGULAR prompt comes up and you are ready to start your SINGULAR session.

It is a good idea to take a look at the (well documented) file `.emacs-singular` in the singular-emacs-home-directory, which comes along with the distribution. In it you find some useful initializations of the SINGULAR interface as well as some lisp code, which, for example, adds a button to the XEmacs toolbar. Some of this code might be useful for your `.emacs` file, too. And if you are an Emacs wizard, it is of course a good idea to take a look at `singular.el` in the singular-emacs-home-directory.

CYGWIN and ESingular

X11 server install `xlaunch`, `emacs-X11`, `xterm` and all dependencies. Create with `xlaunch` a startup file for the X-server which also starts the client `xterm`. From that one can start ESingular.

fork problems The simplest way to overcome fork problem is to run `/usr/bin/rebase-trigger full`, then stop all Cygwin processes and services, and then run `setup-x86.exe`. The _autorebase postinstall script will then take care of the rebase. Occasionally it is necessary to reboot the computer before doing this.

Starting, interrupting and stopping SINGULAR

There are the following commands to start and stop SINGULAR:

- `singular-other` (or menu `Singular`, item `Start...`)

  Starts a SINGULAR process and asks for the following four parameters in the minibuffer area:

  1. The SINGULAR executable. This can either be a file name with complete path, e.g., `/local/bin/Singular`. Then exactly this executable is started. The path may contain the character `~` denoting your home directory. Or it can be the name of a command without path, e.g., `Singular`. Then the executable is searched for in your `$PATH` environment variable.
  2. The default working directory. This is the path to an existing directory, e.g., `~/work`. The current directory is set to this directory before SINGULAR is started.
  3. Command line options. You can set any SINGULAR command line option (see Section 3.1.6 [Command line options], page 19).
  4. The buffer name. You can specify the name of the buffer the interface is running in.

- `singular` (or menu `Singular`, item `Start default`)

  Starts SINGULAR with default settings for the executable, the working directory, command line switches, and the buffer name. You can customize this default settings, see Section 3.2.4 [Customization of the Emacs interface], page 27.

- `singular-exit-singular` (bound to `C-c $` or menu `Singular`, item `Exit`)

  Kills the running SINGULAR process of the current buffer (but does not kill the buffer). Once you have killed a SINGULAR process you can start a new one in the same buffer with the command `singular` (or select the item `Start default` of the `Singular` menu).

- `singular-restart` (bound to `C-c C-r` or menu `Singular`, item `Restart`)

  Kills the running SINGULAR process of the current buffer and starts a new process in the same buffer with exactly the same command line arguments as before.

- `singular-control-c` (bound to `C-c C-c` or menu `Singular`, item `Interrupt`)

  Interrupt the Singular process running in the current buffer. Asks whether to (`a`)bort the current Singular command, (`q`)uit or (`r`)estart the current Singular process, or (`c`)ontinue without doing anything (default).

Whenever a Singular process is started within the Emacs interface, the contents of a special startup file (by default `~/.emacs-singularrc`) is pasted as input to Singular at the very end of the usual startup sequence (see Section 3.1.7 [Startup sequence], page 22). The name of the startup file can be changed, see Section 3.2.4 [Customization of the Emacs interface], page 27.

3.2.3 Demo mode

The Emacs interface can be used to run interactive Singular demonstrations. A demonstration is started by loading a so-called Singular demo file with the Emacs command `singular-demo-load`, bound to `C-c C-d`, or with the menu `Commands->Load Demo`.

A Singular demo file should consist of Singular commands separated by blank lines. When running a demo, the input up to the next blank line is echoed to the screen. Hitting ⟨RETURN⟩ executes the echoed commands and shows their output. Hitting ⟨RETURN⟩ again, echos the next commands to the screen, and so on, until all commands of the demo file are executed. While running a demo, you can execute other commands on the Singular prompt: the next input from the demo file is then echoed again, if you hit ⟨RETURN⟩ on an empty input line.

A Singular demo can prematurely be exited by either starting another demo, or by executing the Emacs command `singular-demo-exit` (menu: `Commands->Exit Demo`).

Some aspects of running Singular demos can be customized. See Section 3.2.4 [Customization of the Emacs interface], page 27, for more info.

3.2.4 Customization of the Emacs interface

Emacs provides a convenient interface to customize the behavior of Emacs and the SINGULAR Emacs interface for your own needs. You enter the customize environment by either calling `M-x customize` (on XEmacs you afterwards have to enter `emacs` in the minibuffer area) or by selecting the menu item `Options->Customize->Emacs...` for XEmacs, and the menu item `Help->Customize->Toplevel Customization Group` for Emacs, resp. A brief introduction to the customization mode comes up with the customization buffer. All customizable parameters are hierarchically grouped and you can browse through all these groups and change the values of the parameters using the mouse. At the end you can safe your settings to a file making your changes permanent.

To change the settings of the Singular Emacs interface you can either select the item `Preferences` of the `Singular` menu, call `M-x customize-group` and give the argument `singular-interactive` in the minibuffer area, or browse from the top-level customization group through the path `External->Singular->Singular interactive`.

The Singular interface customization buffer is divided into four groups:

- Singular Faces

  Here you can specify various faces used if font-lock-mode is enabled (which, by default, is).

- Singular Sections And Foldings

  Here you can specify special faces for Singular input and output and change the text used as replacement for folded sections.

  For doing this, you also might find handy the function `customize-face-at-point`, which lets you customize the face at the current position of point. This function is automatically defined if you run `ESingular`). Otherwise, you should add its definition (see below) to your personal `.emacs` file.

- Singular Interactive Miscellaneous

  Here you can specify various things such as the behavior of the cursor keys, the name of the special SINGULAR startup file, the appearance of the help window, or the default values for the `singular` command.

- Singular Demo Mode

  Here you can specify how chunks of the demo file are divided, or specify a default directory for demo files.

When you run `ESingular`, the settings of customized variables are saved in the file `$HOME/.emacs-singular-cust`. Otherwise, the settings are appended to your `.emacs` file. Among others, this means that the customized settings of `ESingular` are not automatically taken over by a "normal" Emacs, and vice versa.

3.2.5 Editing SINGULAR input files with Emacs

Since SINGULAR's programming language is similar to C, you should use the Emacs C/C++-mode to edit SINGULAR input files and SINGULAR libraries. Among others, this Emacs mode provides automatic indentation, line-breaking and keyword highlighting.

When running `ESingular`, the C/C++-mode is automatically turned on whenever a file with the suffix `.sing`, or `.lib` is loaded.

For Emacs sessions which were not started by `ESingular`, you should add the following to your `.emacs` file:

```
;; turn on c++-mode for files ending in ".sing" and ".lib"
(setq auto-mode-alist (cons '("\\.sing\\'" . c++-mode) auto-mode-alist))
(setq auto-mode-alist (cons '("\\.lib\\'" .  c++-mode) auto-mode-alist))
;; turn-on fontification for c++-mode
(add-hook 'c++-mode-hook
         (function (lambda () (font-lock-mode 1))))
;; turn on aut-new line and hungry-delete
(add-hook 'c++-mode-hook
          (function (lambda () (c-toggle-auto-hungry-state 1))))
;; a handy function for customization
(defun customize-face-at-point ()
  "Customize face which point is at."
  (interactive)
  (let ((face (get-text-property (point) 'face)))
    (if face
        (customize-face face)
      (message "No face defined at point"))))
```

Notice that you can change the default settings for source-code highlighting (colors, fonts, etc.) by customizing the respective faces using the `Customize` feature of Emacs. For doing this, you might find handy the above given function `customize-face-at-point`, which lets you customize the face of the current position of point (this function is automatically defined if you run `ESingular`).

3.2.6 Top 20 Emacs commands

Here is a list of the 20 probably most useful commands when using the SINGULAR Emacs interface.

Starting and stopping of SINGULAR:

- `singular` (menu `Singular->Start Default...`): starts SINGULAR using default arguments.
- `singular-other` (menu `Singular->Start`): starts SINGULAR asking for several arguments in the minibuffer area.
- `singular-exit` (key `C-c $` or menu `Singular->Exit`): kills the SINGULAR process running in the current buffer (but does not kill the buffer).
- `singular-restart` (key `C-c C-r` or menu `Singular->Restart`): kills the SINGULAR process running in the current buffer and starts a new SINGULAR process with exactly the same arguments as before.

Editing input and output:

- `singular-beginning-of-line` (key `C-a`): moves point to beginning of line, then skips past the SINGULAR prompt, if any.
- `singular-toggle-truncate-lines` (key `C-c C-t` or menu `Commands->Truncate lines`): toggles whether long lines should be truncated or not. If lines are not truncated, the commands `singular-scroll-left` and `singular-scroll-right` are useful to scroll left and right, resp.
- `singular-dynamic-complete` (key `TAB`): performs context specific completion. If point is inside a string, file name completion is done. If point is at the end of a help command (i.e., `help` or `?`), completion on SINGULAR help topics is done. If point is at the end of an example command (i.e., `example`), completion is done on SINGULAR examples. In all other cases, completion on SINGULAR commands is done.
- `singular-folding-toggle-fold-latest-output` (key `C-c C-o` or menu `Commands->Fold/Unfold Latest Output`): toggles folding of the latest output section. If your last SINGULAR command produced a huge output, simply type `C-c C-o` and it will be replaced by a single line.
- `singular-folding-toggle-fold-at-point` (key `C-c C-f` or menu `Commands->Fold/Unfold At Point`): toggles folding of the section the point currently is in.
- `singular-folding-fold-all-output` (menu `Commands->Fold All Output`): folds all SINGULAR output, replacing each output section by a single line.
- `singular-folding-unfold-all-output` (menu `Commands->Unfold All Output`): unfolds all SINGULAR output sections showing their true contents.

Loading of files and SINGULAR demo mode:

- `singular-load-library` (key `C-c C-l` or menu `Commands->Libraries->other...`): asks for a standard library name or a library file in the minibuffer (hit `TAB` for completion) and loads the library into SINGULAR. The submenu `Libraries` of the `Commands` menu also provides a separate menu item for each standard library.
- `singular-load-file` (key `C-c <` or menu `Commands->Load File...`): asks for a file name in the minibuffer (which is expanded using `expand-file-name` if given a prefix argument) and loads the file into SINGULAR.
- `singular-demo-load` (key `C-c C-d` or menu `Commands->Load Demo...`): asks for a file name of a SINGULAR demo file in the minibuffer area (hit `SPACE` for completion) and enters the SINGULAR demo mode showing the first chunk of the demo.
- `singular-demo-exit` (menu `Commands->Exit Demo`): exits from SINGULAR demo mode and cleans up everything that is left from the demo.

Help and Customization:

- `singular-help` (key `C-h C-s` or menu `Singular->Singular Help`): asks for a SINGULAR help topic in the minibuffer (hit `TAB` for completion) and shows the help text in a separate buffer.

- `singular-example` (key `C-c C-e` or menu `Singular->Singular Example`): asks for a SINGULAR command in the minibuffer (hit `TAB` for completion) and executes the example of this command in the current SINGULAR buffer.
- `customize-group` (menu `Singular->Preferences`): enters the customization group of the SINGULAR Emacs interface. (If called via `M-x customize-group` give argument `singular-interactive` in the minibuffer area.)

3.3 Rings and orderings

All non-trivial algorithms in SINGULAR require the prior definition of a ring. Such a ring can be

1. a polynomial ring over a field,
2. a polynomial ring over a ring
3. a localization of 1.
4. a quotient ring by an ideal of 1. or 2.,
5. a tensor product of 1. or 2.

Except for quotient rings, all of these rings are realized by choosing a coefficient field, ring variables, and an appropriate global or local monomial ordering on the ring variables. See Section 3.3.3 [Term orderings], page 34, Appendix C [Mathematical background], page 702.

The coefficient field of the rings may be

1. the field of rational numbers Q (`QQ`),
2. finite fields Z/p, p a prime ≤ 2147483647,
3. finite fields $\mathrm{GF}(p^n)$ with p^n elements, p a prime, $p^n \leq 2^{16}$,
4. transcendental extension of Q or Z/p ,
5. simple algebraic extension of Q or Z/p ,
6. the field of real numbers represented by floating point numbers of a user defined precision,
7. the field of complex numbers represented by (pairs of) floating point numbers of a user defined precision,
8. the ring of integers (`ZZ`),
9. finite rings Z/m with $m \in Z$.

In case of coefficient rings, which are not fields, only the following functions are guaranteed to work:

- basic polynomial arithmetic, i.e. addition, multiplication, division, exponentation
- std, i.e. computing standard bases (and related: syz, etc.)
- interred
- reduce

Throughout this manual, the current active ring in SINGULAR is called basering. The reserved name `basering` in SINGULAR is an alias for the current active ring. The basering can be set by declaring a new ring as described in the following subsections or by using the commands `setring` and `keepring`. See Section 5.2.11 [keepring], page 293, Section 5.1.137 [setring], page 255.

Objects of ring dependent types are local to a ring. To access them after a change of the basering they have to be mapped using `map` or by the functions `imap` or `fetch`. See Section 3.5.4 [Objects], page 45, Section 5.1.37 [fetch], page 179, Section 5.1.57 [imap], page 196, Section 4.11 [map], page 104.

All changes of the basering in a procedure are local to this procedure unless a `keepring` command is used as the last statement of the procedure. See Section 3.7 [Procedures], page 51, Section 5.2.11 [keepring], page 293.

3.3.1 Examples of ring declarations

The exact syntax of a ring declaration is given in the next two subsections; this subsection lists some examples first. Note that the chosen ordering implies that a unit-elements of the ring will be among the elements with leading monomial 1. For more information, see Section B.2 [Monomial orderings], page 696.

Every floating point number in a ring consists of two parts, which may be chosen by the user. The leading part represents the number and the rest is for numerical stability. Two numbers with a difference only in the rest will be regarded equal.

- the ring $Z/32003[x, y, z]$ with degree reverse lexicographical ordering. The exact ring declaration may be omitted in the first example since this is the default ring:

```
ring r1;
ring r2 = 32003,(x,y,z),dp;
ring r3=(ZZ/32003)[x,y,z];
ring r4 = (ZZ/32003),(x,y,z),dp;
```

- similar examples with indexed variables. The ring variables of r1 are going to be x(1)..x(10); in r2 they will be x(1)(1), x(1)(2), ..., x(1)(8), x(2)(1), ..., x(5)(8):

```
ring r1 = 32003,(x(1..10)),dp;
ring r2 = 32003,(x(1..5)(1..8)),dp;
ring r3 = (ZZ/32003)[x(1..5)(1..8)];
ring r4 = (ZZ/32003),(x(1..5)(1..8)),dp;
```

- the ring $Q[a, b, c, d]$ with lexicographical ordering:

```
ring r1 = 0,(a,b,c,d),lp;
ring r2 = QQ,(a,b,c,d),lp;
```

- the ring $Z/7[x, y, z]$ with local degree reverse lexicographical ordering. The non-prime 10 is converted to the next lower prime in the second example:

```
ring r1 = 7,(x,y,z),ds;
ring r2 = 10,(x,y,z),ds;
ring r3 = (ZZ/7),(x,y,z),ds;
```

- the ring $Z/7[x_1, \ldots, x_6]$ with lexicographical ordering for x_1, x_2, x_3 and degree reverse lexicographical ordering for x_4, x_5, x_6:

```
ring r1 = 7,(x(1..6)),(lp(3),dp);
ring r2 = (ZZ/7),(x(1..6)),(lp(3),dp);
```

- the localization of $(Q[a, b, c])[x, y, z]$ at the maximal ideal

  (x, y, z) :

```
ring r1 = 0,(x,y,z,a,b,c),(ds(3), dp(3));
ring r2 = QQ,(x,y,z,a,b,c),(ds(3), dp(3));
```

- the ring $Q[x, y, z]$ with weighted reverse lexicographical ordering. The variables x , y , and z have the weights 2, 1, and 3, respectively, and vectors are first ordered by components (in descending order) and then by monomials:

```
ring r1 = 0,(x,y,z),(c,wp(2,1,3));
ring r2 = QQ,(x,y,z),(c,wp(2,1,3));
```

  For ascending component order, the component ordering `C` has to be used.

- the ring $K[x, y, z]$, where $K = Z/7(a, b, c)$ denotes the transcendental extension of $Z/7$ by a , b and c with degree lexicographical ordering:

```
ring r = (7,a,b,c),(x,y,z),Dp;
```

- the ring $K[x,y,z]$, where $K = Z/7[a]$ denotes the algebraic extension of degree 2 of $Z/7$ by a. In other words, K is the finite field with 49 elements. In the first case, a denotes an algebraic element over $Z/7$ with minimal polynomial $\mu_a = a^2 + a + 3$, in the second case, a

  refers to some generator of the cyclic group of units of K :
  ```
  ring r = (7,a),(x,y,z),dp; minpoly = a^2+a+3;
  ring r = (7^2,a),(x,y,z),dp;
  ```
- the ring $R[x,y,z]$, where R denotes the field of real numbers represented by simple precision floating point numbers. This is a special case:
  ```
  ring r = real,(x,y,z),dp;
  ```
- the ring $R[x,y,z]$, where R denotes the field of real numbers represented by floating point numbers of 50 valid decimal digits and the same number of digits for the rest:
  ```
  ring r = (real,50),(x,y,z),dp;
  ```
- the ring $R[x,y,z]$, where R denotes the field of real numbers represented by floating point numbers of 10 valid decimal digits and with 50 digits for the rest:
  ```
  ring r = (real,10,50),(x,y,z),dp;
  ```
- the ring $R(j)[x,y,z]$, where R denotes the field of real numbers represented by floating point numbers of 30 valid decimal digits and the same number for the rest. j denotes the imaginary unit.
  ```
  ring r = (complex,30,j),(x,y,z),dp;
  ```
- the ring $R(i)[x,y,z]$, where R denotes the field of real numbers represented by floating point numbers of 6 valid decimal digits and the same number for the rest. i is the default for the imaginary unit.
  ```
  ring r = complex,(x,y,z),dp;
  ```
- the quotient ring $Z/7[x,y,z]$ modulo the square of the maximal ideal (x,y,z) :
  ```
  ring R = 7,(x,y,z), dp;
  qring r = std(maxideal(2));
  ```
- the ring $Z[x,y,z]$:
  ```
  ring R = integer,(x,y,z), dp;
  ```
- the ring $Z/6^3[x,y,z]$:
  ```
  ring R = (integer, 6, 3),(x,y,z), dp;
  ```
- the ring $Z/100[x,y,z]$:
  ```
  ring R = (integer, 100),(x,y,z), dp;
  ```

3.3.2 General syntax of a ring declaration

Rings

Syntax: `ring` name `=` (coefficients)`,` `(` names_of_ring_variables `)``,` `(` ordering `)``;` or
`ring` name `=` cring `[` names_of_ring_variables `]`

Default: `(ZZ/32003)[x,y,z];`

Purpose: declares a ring and sets it as the current basering. The second form sets the ordering to `(dp,C)`. `cring` stands currently for `QQ` (the rationals), `ZZ` (the integers) or `(ZZ/m)` (the field (m prime and <2147483648) resp. ring of the integers modulo m).

The coefficients (for the first form) are given by one of the following:

1. a `cring` as given above

2. a non-negative int_expression less than 2147483648 (2^31).
   The int_expression should either be 0, specifying the field of rational numbers Q, or a prime number p, specifying the finite field with p elements. If it is not a prime number, int_expression is converted to the next lower prime number.
3. an expression_list of an int_expression and one or more names.
   The int_expression specifies the characteristic of the coefficient field as described above. The names are used as parameters in transcendental or algebraic extensions of the coefficient field. Algebraic extensions are implemented for one parameter only. In this case, a minimal polynomial has to be defined by an assignment to `minpoly`. See Section 5.3.3 [minpoly], page 298.
4. an expression_list of an int_expression and a name.
   The int_expression has to be a prime number p to the power of a positive integer n. This defines the Galois field GF(p^n) with p^n elements, where p^n has to be less than or equal to 2^{15}. The given name refers to a primitive element of GF(p^n) generating the multiplicative group. Due to a different internal representation, the arithmetic operations in these coefficient fields are faster than arithmetic operations in algebraic extensions as described above.
5. an expression_list of the name `real` and two optional int_expressions determining the precision in decimal digits and the size for the stabilizing rest. The default for the rest is the same size as for the representation. An exeption is the name `real` without any integers. These numbers are implemented as machine floating point numbers of single precision. Note that computations over all these fields are not exact.
6. an expression_list of the name `complex`, two optional int_expression and a name. This specifies the field of complex numbers represented by floating point numbers with a precision similar to `real`. An expression_list without int_expression defines a precision and rest with length 6. The name of the imaginary unit is given by the last parameter. Note that computations over these fields are not exact.
7. an expression_list with the name `integer`. This specifies the ring of integers.
8. an expression_list with the name `integer` and one subsequent int_expression. This specifies the ring of integers modulo the given int_expression.
9. an expression_list with the name `integer` and two int_expressions `b` and `e`. This specifies the ring of integers modulo b^e. If `b = 2` and `e < int_bit_size` an optimized implementation is used.

'names_of_ring_variables' is a list of names or indexed names.

'ordering' is a list of block orderings where each block ordering is either

1. `lp`, `dp`, `Dp`, `ls`, `ds`, or `Ds` optionally followed by a size parameter in parentheses.
2. `wp`, `Wp`, `ws`, `Ws`, or `a` followed by a weight vector given as an intvec_expression in parentheses.
3. `M` followed by an intmat_expression in parentheses.
4. `c` or `C`.

For the definition of the orderings, see Section B.2 [Monomial orderings], page 696.

If one of coefficients, names_of_ring_variables, and ordering consists of only one entry, the parentheses around this entry may be omitted.

Quotient rings

Syntax: `qring` name `=` ideal_expression `;`

Default: none

Purpose: declares a quotient ring as the basering modulo ideal_expression, and sets it as current basering.

ideal_expression has to be represented by a standard basis.

The most convenient way to map objects from a ring to its quotient ring and vice versa is to use the `fetch` function (see Section 5.1.37 [fetch], page 179).

Singular computes in a quotient ring as long as possible with the given representative of a polynomial, say, `f`. I.e., it usually does not reduce `f` w.r.t. the quotient ideal. This is only done when necessary during standard bases computations or by an explicit reduction using the command `reduce(f, std(0))` (see Section 5.1.127 [reduce], page 246).

Operations based on standard bases (e.g. `std`,`groebner`, etc., `reduce`) and functions which require a standard basis (e.g. `dim`,`hilb`, etc.) operated with the residue classes; all others on the polynomial objects.

Example:

```
  // defintion and usage:
  ring r=(ZZ/32003)[x,y];
  poly f=x3+yx2+3y+4;
  qring q=std(maxideal(2));
  basering;
↦ // coefficients: ZZ/32003
↦ // number of vars : 2
↦ //        block   1 : ordering dp
↦ //                  : names    x y
↦ //        block   2 : ordering C
↦ // quotient ring from ideal
↦ _[1]=y2
↦ _[2]=xy
↦ _[3]=x2
  poly g=fetch(r, f);
  g;
↦ x3+x2y+3y+4
  reduce(g,std(0));
↦ 3y+4
  // polynomial and residue class:
  ring R=QQ[x,y];
  qring Q=std(y);
  poly p1=x;
  poly p2=x+y;
  // comparing polynomial objects:
  p1==p2;
↦ 0
  // comparing residue classes:
  reduce(p1,std(0))==reduce(p2,std(0));
↦ 1
```

3.3.3 Term orderings

Any polynomial (resp. vector) in Singular is ordered w.r.t. a term ordering (or, monomial ordering), which has to be specified together with the declaration of a ring. Singular stores and displays a polynomial (resp. vector) w.r.t. this ordering, i.e., the greatest monomial (also called the leading monomial) is the first one appearing in the output polynomial, and the smallest monomial is the last one.

Remark: The novice user should generally use the ordering `dp` for computations in the polynomial ring $K[x_1,\ldots,x_n]$, resp. `ds` for computations in the localization $\mathrm{Loc}_{(x)}K[x_1,\ldots,x_n]$). For more details, see Appendix B [Polynomial data], page 695.

In a ring declaration, SINGULAR offers the following orderings (but see also Section B.2 [Monomial orderings], page 696):

1. Global orderings

   `lp` lexicographical ordering

   `rp` reverse lexicographical ordering, i.e. a lexicographical ordering from the right with 1 < x_1 < ... <x_n (should not be used as it reverses the "natural" x_1 > ... > x_n, reorder the variables instead)

   `dp` degree reverse lexicographical ordering

   `Dp` degree lexicographical ordering

   `wp(` intvec_expression `)`
   weighted reverse lexicographical ordering; the weight vector is expected to consist of positive integers only.

   `Wp(` intvec_expression `)`
   weighted lexicographical ordering; the weight vector is expected to consist of positive integers only.

   Global orderings are well-orderings, i.e., $1 < x$ for each ring variable x . They are denoted by a `p` as the second character in their name.

2. Local orderings

   `ls` negative lexicographical ordering

   `rs` negative reverse lexicographical ordering, i.e. a lexicographical ordering from the right (should not be used as it reverses the "natural" x_1 < ... < x_n, reorder the variables instead)

   `ds` negative degree reverse lexicographical ordering

   `Ds` negative degree lexicographical ordering

   `ws(` intvec_expression `)`
   (general) weighted reverse lexicographical ordering; the first element of the weight vector has to be non-zero.

   `Ws(` intvec_expression `)`
   (general) weighted lexicographical ordering; the first element of the weight vector has to be non-zero.

   Local orderings are not well-orderings. They are denoted by an `s` as the second character in their name.

3. Matrix orderings

   `M(` intmat_expression `)`
   intmat_expression has to be an invertible square matrix

   Using matrix orderings, SINGULAR can compute standard bases w.r.t. any monomial ordering which is compatible with the natural semi-group structure on the monomials. In practice, the predefined global and local orderings together with the block orderings should be sufficient in most cases. These orderings are faster than their corresponding matrix orderings since evaluation of a matrix ordering is more time consuming.

4. Extra weight vector

   `a(` intvec_expression `)`
   : an extra weight vector `a(` intvec_expression `)` may precede any monomial ordering

5. Product ordering

   `(` ordering [`(` int_expression `)`], `...` `)`
   : any of the above orderings and the extra weight vector may be combined to yield product or block orderings

   The orderings `lp`, `dp`, `Dp`, `ls`, `ds`, `Ds` and `rp` may be followed by an int_expression in parentheses giving the size of the block. For the last block the size is calculated automatically. For weighted orderings, the size of the block is given by the size of the weight vector. The same holds analogously for matrix orderings.

6. Module orderings

   `(` ordering`, ...,` `C` `)`
   `(` ordering`, ...,` `c` `)`
   : sort polynomial vectors by the monomial ordering first, then by components

   `( C,` ordering`, ...` `)`
   `( c,` ordering`, ...` `)`
   : sort polynomial vectors by components first, then by the monomial ordering

   Here a capital `C` sorts generators in ascending order, i.e., `gen(1) < gen(2) < ...`. A small `c` sorts in descending order, i.e., `gen(1) > gen(2) > ...`. It is not necessary to specify the module ordering explicitly since `(` ordering`, ..., C )` is the default.

   In fact, `c` or `C` may be specified anywhere in a product ordering specification, not only at its beginning or end. All monomial block orderings preceding the component ordering have higher precedence, all monomial block orderings following after it have lower precedence.

For a mathematical description of these orderings, see Appendix B [Polynomial data], page 695.

3.3.4 Coefficient rings

Singular supports coefficient ranges which are not fields, i.e. the integers Z and the finite rings Z/n for a number `n`. These coefficient rings were implemented in Singular 3.0.5 and at the moment only limited functionality is available.

p-adic numbers

The p-adic integers Z_p are the projective limit of the finite rings Z/p^n for `n` to infinity. Therefore, computations in this ring can be approximated by computations in Z/p^n for large `n`.

3.4 Implemented algorithms

The basic algorithm in Singular is a general standard basis algorithm for any monomial ordering which is compatible with the natural semi-group structure of the exponents. This includes well-orderings (Buchberger algorithm to compute a Groebner basis) and tangent cone orderings (Mora algorithm) as special cases.

Nonetheless, there are a lot of other important algorithms:

- Algorithms to compute the standard operations on ideals and modules: intersection, ideal quotient, elimination, etc.

- Different Syzygy algorithms and algorithms to compute free resolutions of modules.
- Combinatorial algorithms to compute dimensions, Hilbert series, multiplicities, etc.
- Algorithms for univariate and multivariate polynomial factorization, resultant and gcd computations.

Commands to compute standard bases

`facstd` Section 5.1.33 [facstd], page 176
computes a list of Groebner bases via the Factorizing Groebner Basis Algorithm, i.e., their intersection has the same radical as the original ideal. It need not be a Groebner basis of the given ideal.

The intersection of the zero-sets is the zero-set of the given ideal.

`fglm` Section 5.1.38 [fglm], page 180
computes a Groebner basis provided that a reduced Groebner basis w.r.t. another ordering is given.

Implements the so-called FGLM (Faugere, Gianni, Lazard, Mora) algorithm. The given ideal must be zero-dimensional.

`groebner` [groebner], page 721
computes a standard resp. Groebner basis using a heuristically chosen method.

This is the preferred method to compute a standard resp. Groebner bases.

`mstd` Section 5.1.97 [mstd], page 223
computes a standard basis and a minimal set of generators.

`std` Section 5.1.147 [std], page 266
computes a standard resp. Groebner basis.

`stdfglm` [stdfglm], page 721
computes a Groebner basis in a ring with a “difficult” ordering (e.g., lexicographical) via `std` w.r.t. a “simple” ordering and `fglm`.

The given ideal must be zero-dimensional.

`stdhilb` [stdhilb], page 721
computes a Groebner basis in a ring with a “difficult” ordering (e.g., lexicographical) via `std` w.r.t. a “simple” ordering and a `std` computation guided by the Hilbert series.

Further processing of standard bases

The next commands require the input to be a standard basis.

`degree` Section 5.1.19 [degree], page 167
computes the (Krull) dimension, codimension and the multiplicity.

The result is only displayed on the screen.

`dim` Section 5.1.24 [dim], page 170
computes the dimension of the ideal resp. module.

`highcorner`
Section 5.1.53 [highcorner], page 192
computes the smallest monomial not contained in the ideal resp. module. The ideal resp. module has to be finite dimensional as a vector space over the ground field.

`hilb` Section 5.1.54 [hilb], page 193
computes the first, and resp. or, second Hilbert series of an ideal resp. module.

`kbase` Section 5.1.67 [kbase], page 203
computes a vector space basis (consisting of monomials) of the quotient of a ring by an ideal resp. of a free module by a submodule.

The ideal resp. module has to be finite dimensional as a vector space over the ground field and has to be represented by a standard basis w.r.t. the ring ordering.

`mult` Section 5.1.98 [mult], page 224
computes the degree of the monomial ideal resp. module generated by the leading monomials of the input.

`reduce` Section 5.1.127 [reduce], page 246
reduces a polynomial, vector, ideal or module to its normal form with respect to an ideal or module represented by a standard basis.

`vdim` Section 5.1.163 [vdim], page 280
computes the vector space dimension of a ring (resp. free module) modulo an ideal (resp. module).

Commands to compute resolutions

`res` [res], page 721
computes a free resolution of an ideal or module using a heuristically chosen method. This is the preferred method to compute free resolutions of ideals or modules.

`fres` Section 5.1.46 [fres], page 185
improved version of Section 5.1.145 [sres], page 264, computes a free resolution of an ideal or module using Schreyer's method. The input has to be a standard basis.

`lres` Section 5.1.81 [lres], page 212
computes a free resolution of an ideal or module with LaScala's method. The input needs to be homogeneous.

`mres` Section 5.1.96 [mres], page 223
computes a minimal free resolution of an ideal or module with the Syzygy method.

`sres` Section 5.1.145 [sres], page 264
computes a free resolution of an ideal or module with Schreyer's method. The input has to be a standard basis.

`nres` Section 5.1.103 [nres], page 228
computes a free resolution of an ideal or module with the standard basis method.

`syz` Section 5.1.152 [syz], page 275
computes the first Syzygy (i.e., the module of relations of the given generators).

Further processing of resolutions

`betti` Section 5.1.4 [betti], page 157
computes the graded Betti numbers of a module from a free resolution.

`minres` Section 5.1.91 [minres], page 220
minimizes a free resolution of an ideal or module.

`regularity`
Section 5.1.128 [regularity], page 247
computes the regularity of a homogeneous ideal resp. module from a given minimal free resolution.

Processing of polynomials

`char_series`
Section 5.1.6 [char_series], page 159
computes characteristic sets of polynomial ideals.

`extgcd` Section 5.1.32 [extgcd], page 176
computes the extended gcd of two polynomials.

This is implemented as extended Euclidean Algorithm, and applicable for univariate polynomials only.

`factorize`
Section 5.1.35 [factorize], page 178
computes factorization of univariate and multivariate polynomials into irreducible factors.

The most basic algorithm is univariate factorization in prime characteristic. The Cantor-Zassenhaus Algorithm is used in this case. For characteristic 0, a univariate Hensel-lifting is done to lift from prime characteristic to characteristic 0. For multivariate factorization in any characteristic, the problem is reduced to the univariate case first, then a multivariate Hensel-lifting is used to lift the univariate factorization.

Factorization of polynomials over algebraic extensions is provided by factoring the norm for univariate polynomials f (the gcd of f and the factors of the norm is a factorization of f) resp. by the extended Zassenhaus algorithm for multivariate polynomials.

`gcd` Section 5.1.48 [gcd], page 187
computes greatest common divisors of univariate and multivariate polynomials.

In the univariate case NTL is used. For prime characteristic, a subresultant gcd is used. In characteristic 0, the EZGCD is used, except for a special case where a modular algorithm is used.

`resultant`
Section 5.1.132 [resultant], page 250
computes the resultant of two univariate polynomials using the subresultant algorithm.

Multivariate polynomials are considered as univariate polynomials in the main variable (which has to be specified by the user).

`vandermonde`
Section 5.1.159 [vandermonde], page 279
interpolates a polynomial from its values at several points

Matrix computations

`bareiss` Section 5.1.3 [bareiss], page 156
implements sparse Gauss-Bareiss method for elimination (matrix triangularization) in arbitrary integral domains.

`det` Section 5.1.22 [det], page 169
computes the determinant of a square matrix.

For matrices with integer entries a modular algorithm is used. For other domains the Gauss-Bareiss method is used.

`minor` Section 5.1.90 [minor], page 218
computes all minors (=subdeterminants) of a given size for a matrix.

Numeric computations

`laguerre` Section 5.1.72 [laguerre], page 206
computes all (complex) roots of a univariate polynomial

`uressolve`
Section 5.1.158 [uressolve], page 278
finds all roots of a 0-dimensional ideal with multivariate resultants

Controlling computations

`option` Section 5.1.108 [option], page 230
allows setting of options for manipulating the behaviour of computations (such as reduction strategies) and for showing protocol information indicating the progress of a computation.

3.5 The SINGULAR language

Singular interprets commands given interactively on the command line as well as given in the context of user-defined procedures. In fact, Singular makes no distinction between these two cases. Thus, Singular offers a powerful programming language as well as an easy-to-use command line interface without differences in syntax or semantics.

In the following, the basic language concepts such as commands, expressions, names, objects, etc., are discussed. See Section 3.7 [Procedures], page 51, and Section 3.8 [Libraries], page 55, for the concepts of procedures and libraries.

In many aspects, the Singular language is similar to the C programming language. For a description of some of the subtle differences, see Section 6.3 [Major differences to the C programming language], page 304.

Elements of the language

The major building blocks of the Singular language are expressions, commands, and control structures. The notion of expressions in the Singular and the C programming language are identical, whereas the notion of commands and control structures only roughly corresponds to C statements.

- An "expression" is a sequence of operators, functions, and operands that specifies a computation. An expression always results in a value of a specific type. See Chapter 4 [Data types], page 73, and its subsections (e.g., Section 4.16.2 [poly expressions], page 119), for information on how to build expressions.
- A "command" is either a declaration, an assignment, a call to a function without return value, or a print command. For detailed information, see Section 3.5.1 [General command syntax], page 41.
- "Control structures" determine the execution sequence of commands. Singular provides control structures for conditional execution (`if ... else`) and iteration (`for` and `while`). Commands may be grouped in pairs of `{ }` (curly brackets) to form blocks. See Section 5.2 [Control structures], page 284, for more information.

Other notational conventions

For user-defined functions, the notions of "procedure" and "function" are synonymous.

As already mentioned above, functions without return values are called commands. Furthermore, whenever convenient, the term "command" is used for a function, even if it does return a value.

3.5.1 General command syntax

In SINGULAR a command is either a declaration, an assignment, a call to a function without return value, or a print command. The general form of a command is described in the following subsections.

Declaration

1. type name `=` expression `;`
   declares a variable with the given name of the given type and assigns the expression as initial value to it. Expression is an expression of the specified type or one that can be converted to that type. See Section 3.5.5 [Type conversion and casting], page 45.
2. `alias` type name
   Introduces name as an alternative, read-only name for another variable_name. Can only be used in procedure headings to avoid copying large data.
3. type name_list `=` expression_list `;`
   declares variables with the given names and assigns successively each expression of expression_list to the corresponding name of name_list. Both lists must be of the same length. Each expression in expression_list is an expression of the specified type or one that can be converted to that type. See Section 3.5.5 [Type conversion and casting], page 45.
4. type name `;`
   declares a variable with the given name of the given type and assigns the default value of the specific type to it.

See Section 3.5.3 [Names], page 44, for more information on declarations. See Chapter 4 [Data types], page 73, for a description of all data types known to SINGULAR.

```
ring r;                    // the default ring
poly f,g = x^2+y^3,xy+z2; // the polynomials f=x^2+y^3 and g=x*y+z^2
ideal I = f,g;             // the ideal generated by f and g
matrix m[3][3];            // a 3 x 3 zero matrix
int i=2;                   // the integer i=2
```

Assignment

4. name `=` expression `;`
   assigns expression to name.
5. name_list `=` expression_list `;`
   assigns successively each expression of expression_list to the corresponding name of name_list. Both lists must be of the same length. This is not a simultaneous assignment. Thus, `f, g = g, f;` does not swap the values of `f` and `g`, but rather assigns `g` to both `f` and `g`.

A type conversion of the type of expression to the type of name must be possible. See Section 3.5.5 [Type conversion and casting], page 45.

An assignment itself does not yield a value. Hence, compound assignments like `i = j = k;` are not allowed and result in an error.

```
f = x^2 + y^2 ;      // overrides the old value of f
I = jacob(f);
f,g = I[1],x^2+y^2 ; // overrides the old values of f and g
```

Function without return value

6. function_name [(argument_list)] ;
   calls function function_name with arguments argument_list.

The function may have output (not to be confused with a return value of type string). See Section 5.1 [Functions], page 154. Functions without a return value are specified there to have a return type 'none'.

Some of these functions have to be called without parentheses, e.g., `help`, `LIB`.

```
  ring r;
  ideal i=x2+y2,x;
  i=std(i);
  degree(i);        // degree has no return value but prints output
↦ // dimension (proj.)  = 0
↦ // degree (proj.)   = 2
```

Print command

7. expression ;
   prints the value of an expression, for example, of a variable.

Use the function `print` (or the procedure `show` from inout.lib) to get a pretty output of various data types, e.g., matrix or intmat. See Section 5.1.117 [print], page 238.

```
  int i=2;
  i;
↦ 2
  intmat m[2][2]=1,7,10,0;
  print(m);
↦      1     7
↦     10     0
```

3.5.2 Special characters

The following characters and operators have special meanings:

`=` assignment

`{, }` parentheses for block programming

`(, )` in expressions, for indexed names and for argument lists

`[, ]` access operator for strings, integer vectors, ideals, matrices, polynomials, resolutions, and lists. Used to build vectors of polynomials. Example: `s[3]`, `m[1,3]`, `i[1..3]`, `[f,g+x,0,0,1]`.

`+` addition operator

`++` increment operator

`-` subtraction operator

`--` decrement operator

`*` multiplication operator

`/` division operator. See Section 6.4 [Miscellaneous oddities], page 308, for the difference between the division operators `/` and `div`.

`%` modulo operator (`mod` is an alias to `%`): result is always non-negative

`^` or `**` exponentiation operator

`==` comparison operator equal

`!=` or `<>` comparison operator not equal

`>=` comparison operator larger than or equal to

`>` comparison operator larger

`<=` comparison operator smaller than or equal to

`<` comparison operator smaller. Also used for file input. See Section 5.1.40 [filecmd], page 182.

`!` boolean operator not

`&&` boolean operator and

`||` boolean operator or

`"` delimiter for string constants

`` ` `` delimiter for name substitution

`?` synonym for `help`

`//` comment delimiter. Comment extends to the end of the line.

`/*` comment delimiter. Starts a comment which ends with `*/`.

`*/` comment delimiter. Ends a comment which starts with `/*`.

`;` statement separator

`,` separator for expression lists and function arguments

`\` escape character for `"` and `\` within strings

`..` interval specifier returning intvec. E.g., `1..3` which is equivalent to the intvec `1, 2, 3`.

`:` repeated entry. E.g., `3:5` generates an intvec of length 5 with constant entries 3, i.e., (3, 3, 3, 3, 3).

`::` accessor for package members. E.g., `MyPackage::i` accesses variable `i` in package `MyPackage`.

`_` value of expression displayed last

`~` breakpoint in procedures

`#` list of parameters in procedures without explicit parameter list

`$` terminates Singular

3.5.3 Names

SINGULAR is a strongly typed language. This means that all names (= identifiers) have to be declared prior to their use. For the general syntax of a declaration, see the description of declaration commands (see Section 3.5.1 [General command syntax], page 41).

See Chapter 4 [Data types], page 73, for a description of SINGULAR's data types. See Section 5.1.156 [typeof], page 277, for a short overview of possible types. To get information on a name and the object named by it, the `type` command may be used (see Section 5.1.155 [type], page 277).

It is possible to redefine an already existing name if doing so does not change its type. A redefinition first sets the variable to the default value and then computes the expression. The difference between redefining and overriding a variable is shown in the following example:

```
  int i=3;
  i=i+1;        // overriding
  i;
↦ 4
  int i=i+1;    // redefinition
↦ // ** redefining i (  int i=i+1;    // redefinition)
  i;
↦ 1
```

User defined names should start with a letter and consist of letters and digits only. As an exception to this rule, the characters `@`, and `_` may be used as part of a name, too (`@` as the first letter is reserved for purposes of library routines). Capital and small letters are distinguished. Indexed names are built as a name followed by an int_expression in parentheses. A list of indexed names can be built as a name followed by an intvec_expression in parentheses. For multi-indices, append an int_expression in parentheses to an indexed name. An alternative multi-index construction is `name_prefix(` index_1, index_2,... `)` where the `name_prefix` must be an undefined name.

```
  ring R;
  int n=3;
  ideal j(3);
  ideal j(n);     // is equivalent to the above
↦ // ** redefining j(3) (  ideal j(n);     // is equivalent to the above)
  ideal j(2)=x;
  j(2..3);
↦ j(2)[1]=x j(3)[1]=0
  ring r=0,(x(1..2)(1..3)(1..2)),dp;
  r;
↦ // coefficients: QQ
↦ // number of vars : 12
↦ //        block   1 : ordering dp
↦ //                  : names    x(1)(1)(1) x(1)(1)(2) x(1)(2)(1) x(1)(2)(2\
  ) x(1)(3)(1) x(1)(3)(2) x(2)(1)(1) x(2)(1)(2) x(2)(2)(1) x(2)(2)(2) x(2)(\
  3)(1) x(2)(3)(2)
↦ //        block   2 : ordering C
  int i(1,2),i(2,3);
  i(2,3);
↦ 0
```

Names must not coincide with reserved names (keywords). Type `reservedName();` to get a list of the reserved names. See Section 5.1.131 [reservedName], page 250. Names should not interfere with names of ring variables or, more generally, with monomials. See Section 6.5 [Identifier resolution], page 310.

The command `listvar` provides a list of the names in use (see Section 5.1.80 [listvar], page 210).

The most recently printed expression is available under the special name _, e.g.,

```
  ring r;
  ideal i=x2+y3,y3+z4;
  std(i);
↦ _[1]=y3+x2
↦ _[2]=z4-x2
  ideal k=_;
  k*k+x;
↦ _[1]=y6+2x2y3+x4
↦ _[2]=y3z4+x2z4-x2y3-x4
↦ _[3]=z8-2x2z4+x4
↦ _[4]=x
  size(_[3]);
↦ 3
```

A string_expression enclosed in `` `...` `` (back ticks) evaluates to the value of the variable given by the string_expression. This feature is referred to as name substitution.

```
  int foo(1)=42;
  string bar="foo";
  `bar+"(1)"`;
↦ 42
```

3.5.4 Objects

Every object in Singular has a type and a value. In most cases it has also a name and in some cases an attribute list. The value of an object may be examined simply by printing it with a print command: object;. The type of an object may be determined by means of the `typeof` function, the attributes by means of the `attrib` function (Section 5.1.156 [typeof], page 277, Section 5.1.2 [attrib], page 154):

```
  ring r=0,x,dp;
  typeof(10);
↦ int
  typeof(10000000000000000);
↦ bigint
  typeof(r);
↦ ring
  attrib(x);
↦ no attributes
  attrib(std(ideal(x)));
↦ attr:isSB, type int
```

Each object of type `poly`, `ideal`, `vector`, `module`, `map`, `matrix`, `number`, or `resolution` belongs to a specific ring. This is also true for `list`, if at least one of the objects contained in the list belongs to a ring. These objects are local to the ring. Their names can be duplicated for other objects in other rings. Objects from one ring can be mapped to another ring using maps or the commands `fetch` or `imap`. See Section 4.11 [map], page 104, Section 5.1.37 [fetch], page 179, Section 5.1.57 [imap], page 196.

All other types do not belong to a ring and can be accessed within every ring and across rings. They can be declared even if there is no active basering.

3.5.5 Type conversion and casting

Type conversion

Assignments convert the type of the right-hand side to the type of the left-hand side of the assignment, if possible. Operators and functions which require certain types of operands can also implicitly convert the type of an expression. It is, for example, possible to multiply a polynomial by an integer because the integer is automatically converted to a polynomial. Type conversions do not act transitively. Possible conversions are:

1.	`intvec`	$\mapsto$ `intmat`
2.	`poly`	$\mapsto$ `ideal`
3.	`bigint`	$\mapsto$ `ideal`
4.	`int`	$\mapsto$ `ideal`
5.	`intmat`	$\mapsto$ `matrix`
6.	`ideal`	$\mapsto$ `matrix`
7.	`module`	$\mapsto$ `matrix`
8.	`number`	$\mapsto$ `matrix`
9.	`poly`	$\mapsto$ `matrix`
10.	`vector`	$\mapsto$ `matrix`
11.	`bigint`	$\mapsto$ `matrix`
12.	`int`	$\mapsto$ `matrix`
13.	`intvec`	$\mapsto$ `matrix`
14.	`ideal`	$\mapsto$ `module`
15.	`matrix`	$\mapsto$ `module`
16.	`vector`	$\mapsto$ `module`
17.	`bigint`	$\mapsto$ `number`
18.	`int`	$\mapsto$ `number`
19.	`number`	$\mapsto$ `poly`
20.	`bigint`	$\mapsto$ `poly`
21.	`int`	$\mapsto$ `poly`
22.	`list`	$\mapsto$ `resolution`
23.	`poly`	$\mapsto$ `vector` (p $\mapsto$ p*`gen(1)`)
24.	`bigint`	$\mapsto$ `vector`
25.	`int`	$\mapsto$ `vector` (i $\mapsto$ i*`gen(1)`)
26.	`int`	$\mapsto$ `bigint`
27.	`int`	$\mapsto$ `intvec`
28.	`string`	$\mapsto$ `link`
29.	`resolution`	$\mapsto$ `list`

Type casting

An expression can be casted to another type by using a type cast expression:
type `(` expression `)`.

Possible type casts are:

to	from
`bigint`	expression `int`, `number`, `poly`
`ideal`	expression lists of `int`, `number`, `poly`
`ideal`	`int`, `matrix`, `module`, `number`, `poly`, `vector`
`int`	`number`, `poly`
`intvec`	expression lists of `int`, `intmat`
`intmat`	`intvec` (see Section 4.7.3 [intmat type cast], page 90)
`list`	expression lists of any type

`matrix`	`module`, `ideal`, `vector`, `matrix`. There are two forms to convert something to a matrix: if `matrix(` expression `)` is used then the size of the matrix is determined by the size of expression. But `matrix(` expression `,` m `,` n `)` may also be used - the result is a $m \times n$ matrix (see Section 4.12.3 [matrix type cast], page 108)
`module`	expression lists of `int`, `number`, `poly`, `vector`
`module`	`ideal`, `matrix`, `vector`
`number`	`poly`
`poly`	`int`, `number`
`ring`	`list` (the inverse of `ringlist`)
`string`	any type (see Section 4.20.3 [string type cast], page 129)

Example:

```
  ring r=0,x,(c,dp);
  number(3x);
↦ 0
  number(poly(3));
↦ 3
  ideal i=1,2,3,4,5,6;
  print(matrix(i));
↦ 1,2,3,4,5,6
  print(matrix(i,3,2));
↦ 1,2,
↦ 3,4,
↦ 5,6
  vector v=[1,2];
  print(matrix(v));
↦ 1,
↦ 2
  module(matrix(i,3,2));
↦ _[1]=[1,3,5]
↦ _[2]=[2,4,6]
  // generators are columns of a matrix
```

3.5.6 Flow control

A block is a sequence of commands surrounded by `{` and `}`.

```
{
  command;
  ...
}
```

Blocks are used whenever SINGULAR is used as a structured programming language. The `if` and `else` structures allow conditional execution of blocks (see Section 5.2.9 [if], page 290, Section 5.2.5 [else], page 286). `for` and `while` loops are available for a repeated execution of blocks (see Section 5.2.8 [for], page 290, Section 5.2.15 [while], page 296). In procedure definitions, the main part and the example section are blocks as well(see Section 4.17 [proc], page 122).

3.6 Input and output

SINGULAR's input and output (short, I/O) are realized using links. Links are the communication channels of SINGULAR, i.e., something SINGULAR can write to and read from. In this section, a short overview of the usage of links and of the different link types is given.

For loading of libraries, see Section 5.1.77 [LIB], page 208. For executing program scripts, see Section 5.1.40 [filecmd], page 182.

Monitoring

A special form of I/O is monitoring. When monitoring is enabled, Singular makes a typescript of everything printed on your terminal to a file. This is useful to create a protocol of a Singular session. The `monitor` command enables and disables this feature (see Section 5.1.93 [monitor], page 221).

How to use links

Recall that links are the communication channels of Singular, i.e., something Singular can write to and read from using the functions `write` and `read`. There are furthermore the functions `dump` and `getdump` which store resp. retrieve the content of an entire Singular session to, resp. from, a link. The `dump` and `getdump` commands are not available for DBM links.

For more information, see Section 5.1.169 [write], page 283, Section 5.1.126 [read], page 245, Section 5.1.26 [dump], page 172, Section 5.1.50 [getdump], page 188.

Example:

```
  ring r; poly p = x+y;
  dump(":w test.sv");   // dump the session to the file test.sv
  kill r;               // kill the basering
  listvar();            // no output after killing the ring
  getdump(":r test.sv");// read the dump from the file
↦ // ** redefining stdfglm (load("standard.lib","try");)
↦ // ** redefining stdfglm (load("standard.lib","try");)
↦ // ** redefining stdhilb (load("standard.lib","try");)
↦ // ** redefining stdhilb (load("standard.lib","try");)
↦ // ** redefining quotientList (load("standard.lib","try");)
↦ // ** redefining quotientList (load("standard.lib","try");)
↦ // ** redefining par2varRing (load("standard.lib","try");)
↦ // ** redefining par2varRing (load("standard.lib","try");)
↦ // ** redefining hilbRing (load("standard.lib","try");)
↦ // ** redefining hilbRing (load("standard.lib","try");)
↦ // ** redefining qslimgb (load("standard.lib","try");)
↦ // ** redefining qslimgb (load("standard.lib","try");)
↦ // ** redefining groebner (load("standard.lib","try");)
↦ // ** redefining groebner (load("standard.lib","try");)
↦ // ** redefining res (load("standard.lib","try");)
↦ // ** redefining res (load("standard.lib","try");)
↦ // ** redefining quot (load("standard.lib","try");)
↦ // ** redefining quot (load("standard.lib","try");)
↦ // ** redefining quot1 (load("standard.lib","try");)
↦ // ** redefining quotient0 (load("standard.lib","try");)
↦ // ** redefining quotient1 (load("standard.lib","try");)
↦ // ** redefining quotient1 (load("standard.lib","try");)
↦ // ** redefining quotient2 (load("standard.lib","try");)
↦ // ** redefining quotient2 (load("standard.lib","try");)
↦ // ** redefining quotient3 (load("standard.lib","try");)
↦ // ** redefining quotient3 (load("standard.lib","try");)
↦ // ** redefining quotient5 (load("standard.lib","try");)
↦ // ** redefining quotient5 (load("standard.lib","try");)
```

```
↦ // ** redefining quotient4 (load("standard.lib","try");)
↦ // ** redefining quotient4 (load("standard.lib","try");)
↦ // ** redefining intersect1 (load("standard.lib","try");)
↦ // ** redefining sprintf (load("standard.lib","try");)
↦ // ** redefining sprintf (load("standard.lib","try");)
↦ // ** redefining printf (load("standard.lib","try");)
↦ // ** redefining printf (load("standard.lib","try");)
↦ // ** redefining fprintf (load("standard.lib","try");)
↦ // ** redefining fprintf (load("standard.lib","try");)
↦ // ** redefining weightKB (load("standard.lib","try");)
↦ // ** redefining weightKB (load("standard.lib","try");)
↦ // ** redefining datetime (load("standard.lib","try");)
↦ // ** redefining datetime (load("standard.lib","try");)
↦ // ** redefining kbclass (load("standard.lib","try");)
↦ // ** redefining delkbclass (load("standard.lib","try");)
↦ // ** redefining kbprepare (load("standard.lib","try");)
↦ // ** redefining checkww (load("standard.lib","try");)
↦ // ** redefining widkbase (load("standard.lib","try");)
↦ // ** redefining weightmon (load("standard.lib","try");)
↦ // ** redefining max (load("standard.lib","try");)
↦ // ** redefining max (load("standard.lib","try");)
↦ // ** redefining min (load("standard.lib","try");)
↦ // ** redefining min (load("standard.lib","try");)
↦ // ** redefining mod_init (load("standard.lib","try");)
  listvar();
↦ // r                              [0]  *ring
↦ //     p                              [0]  poly
```

Specifying a link can be as easy as specifying a filename as a string. Links do not even need to be explicitly opened or closed before, resp. after, they are used. To explicitly open or close a link, the `open`, resp. `close`, commands may be used (see Section 5.1.107 [open], page 230, Section 5.1.10 [close], page 161).

Links have various properties which can be queried using the `status` function (see Section 5.1.146 [status], page 265).

Example:

```
  link l = "ssi:fork";
  l;
↦ // type : ssi
↦ // mode : fork
↦ // name :
↦ // open : no
↦ // read : not open
↦ // write: not open
  open(l);
  status(l, "open");
↦ yes
  close(l);
  status(l, "open");
↦ no
```

ASCII links

Data that can be converted to a string can be written into files for storage or communication with other programs. The data are written in plain ASCII format. Reading from an ASCII link returns a string — conversion into other data is up to the user. This can be done, for example, using the command `execute` (see Section 5.1.31 [execute], page 175).

ASCII links should primarily be used for storing small amounts of data, especially if it might become necessary to manually inspect or manipulate the data.

See Section 4.9.4 [ASCII links], page 96, for more information.

Example:

```
  // (over)write file test.ascii, link is specified as string
  write(":w test.ascii", "int i =", 3, ";");
  // reading simply returns the string
  read("test.ascii");
↦ int i =
↦ 3
↦ ;
↦
  // but now test.ascii is "executed"
  execute(read("test.ascii"));
  i;
↦ 3
```

Ssi links

Data is communicated with other processes (e.g., Singular processes) which may run on the same computer or on different ones. Data exchange is accomplished using TCP/IP links in the ssi format. Reading from an ssi link returns the written expressions (i.e., not a string, in general).

Ssi links should primarily be used for communicating with other programs or for parallel computations (see, for example, Section A.1.8 [Parallelization with ssi links], page 636).

See Section 4.9.5 [Ssi links], page 97, for more information.

Example:

```
  ring r;
  link l = "ssi:tcp localhost:"+system("Singular"); // declare a link explicitly
  open(l);  // needs an open, launches another SINGULAR as a server
  write(l, x+y);
  kill r;
  def p = read(l);
  typeof(p); p;
↦ poly
↦ x+y
  close(l); // shuts down SINGULAR server
```

DBM links

Data is stored in and accessed from a data base. Writing is accomplished by a key and a value and associates the value with the key in the specified data base. Reading is accomplished w.r.t. a key, the value associated to it is returned. Both the key and the value have to be specified as strings. Hence, DBM links may be used only for data which may be converted to or from strings.

DBM links should primarily be used when data needs to be accessed not in a sequential way (like with files) but in an associative way (like with data bases).

See Section 4.9.7 [DBM links], page 100, for more information.

Example:

```
ring r;
// associate "x+y" with "mykey"
write("DBM:w test.dbm", "mykey", string(x+y));
// get from data base what is stored under "mykey"
execute(read("DBM: test.dbm", "mykey"));
↦ x+y
```

3.7 Procedures

Procedures contain sequences of commands in the SINGULAR language. They are used to extend the set of commands by user defined commands. In a SINGULAR session, procedures are defined by either typing them on the command line or by loading them from a library file with the `LIB` or `load` command (see Section 3.8 [Libraries], page 55). A procedure is invoked like normal built-in commands, i.e., by typing its name followed by the list of arguments in parentheses. The invocation then executes the sequence of commands constituting the procedure. All procedures defined in a SINGULAR session can be displayed by entering `listvar(proc);` .

See also See Section 3.8.6 [Procedures in a library], page 58.

3.7.1 Procedure definition

Syntax: [`static`] `proc` proc_name [(<parameter_list>)]
[<help_string>]
`{`
<procedure_body>
`}`
[`example`
`{`
<sequence_of_commands>
`}`]

Purpose:

- Defines a new function, the `proc` proc_name.
- The help string, the parameter list, and the example section are optional. They are, however, mandatory for the procedures listed in the header of a library. The help string is ignored and no example section is allowed if the procedure is defined interactively, i.e., if it is not loaded from a file by the `LIB` or `load` command (see Section 5.1.77 [LIB], page 208 and see Section 5.2.12 [load], page 293).
- Once loaded from a file into a SINGULAR session, the information provided in the help string will be displayed upon entering `help proc_name;`, while the `example` section will be executed upon entering `example proc_name;`. See Section 3.7.2 [Parameter list], page 53, Section 3.7.3 [Help string], page 54, and the example in Section 3.8.6 [Procedures in a library], page 58.
- In the body of a library, each procedure not meant to be accessible by users should be declared static. See Section 3.8.6 [Procedures in a library], page 58.

Example of an interactive procedure definition and its execution:

```
  proc milnor_number (poly p)
  {
    ideal i= std(jacob(p));
    int m_nr=vdim(i);
    if (m_nr<0)
    {
      "// not an isolated singularity";
    }
    return(m_nr);         // the value of m_nr is returned
  }
  ring r1=0,(x,y,z),ds;
  poly p=x^2+y^2+z^5;
  milnor_number(p);
↦ 4
```

Example of a procedure definition in a library:

First, we define the library (and store it as `sample.lib`):

```
// Example of a user accessible procedure
proc tab (int n)
"USAGE:    tab(n);  n integer
RETURNS:  string of n space tabs
EXAMPLE:  example tab; shows an example"
{ return(internal_tab(n)); }
example
{
  "EXAMPLE:"; echo=2;
  for(int n=0; n<=4; n=n+1)
  { tab(4-n)+"*"+tab(n)+"+"+tab(n)+"*"; }
}

// Example of a static procedure
static proc internal_tab (int n)
{ return(" "[1,n]); }
```

Now, we load the library and execute its procedures:

```
  LIB "sample.lib";        // load the library sample.lib
  example tab;             // show an example
↦ // proc tab from lib sample.lib
↦ EXAMPLE:
↦   for(int n=0; n<=4; n=n+1)
↦   { tab(4-n)+"*"+tab(n)+"+"+tab(n)+"*"; }
↦     *+*
↦    * + *
↦   *  +  *
↦  *   +   *
↦ *    +    *
↦ 
  "*"+tab(3)+"*";          // use the procedure tab
↦ *   *
  // the static procedure internal_tab is not accessible
  "*"+internal_tab(3)+"*";
↦    ? `internal_tab(3)` is not defined
```

```
↦    ? error occurred in or before ./examples/Example_of_a_procedure_defini\
   tion_in_a_library:.sing line 5: `  "*"+internal_tab(3)+"*";`
  // show the help section for tab
  help tab;
↦ // ** Could not get 'IdxFile'.
↦ // ** Either set environment variable 'SINGULAR_IDX_FILE' to 'IdxFile',
↦ // ** or make sure that 'IdxFile' is at "%D/singular/singular.idx"
↦ // ** Displaying help in browser 'dummy'.
↦ // ** Use 'system("--browser", <browser>);' to change browser,
↦ // ** where <browser> can be: "dummy", "emacs".
↦    ? No functioning help browser available.
↦    ? error occurred in or before ./examples/Example_of_a_procedure_defini\
   tion_in_a_library:.sing line 7: `  help tab;`
```

3.7.2 Parameter list

Syntax: ( )
(parameter_definition)

Purpose:

- Defines the number, type and names of the arguments of a procedure.
- The parameter_list is optional.
- Adding `list #` as argument to a parameter list means to allow optional parameters. Furthermore, `(list #)` is the default for a parameter list (in case no list is explicitly given). Inside the procedure body, the arguments of `list #` are referenced by `#[1]`, `#[2]`, etc.
- If a procedure has optional parameters, the attribute `default_arg` gives the default values for the optional arguments. This provides in particular the possibility to also change the behaviour of all procedures nested inside the given procedure.

Example:

```
proc x0
{
    // can be called with
... // any number of arguments of any type: #[1], #[2],...
    // number of arguments: size(#)
}

proc x1 ()
{
... // can only be called without arguments
}

proc x2 (ideal i, int j)
{
... // can only be called with 2 arguments,
    // which can be converted to ideal resp. int
}

proc x3 (i,j)
{
... // can only be called with 2 arguments
```

```
    // of any type
    // (i,j) is the same as (def i,def j)
}

proc x5 (i,list #)
{
... // can only be called with at least 1 argument
    // number of arguments: size(#)+1
}

attrib(x5,"default_arg",3);
x5(2); // is equivalent to
x5(2,3);
```

Note:

The parameter_list may stretch across multiple lines.
A parameter may have any type (including the types `proc` and `ring`).
If a parameter is of type ring, then it can only be specified by name, but not with a type. For instance:

```
proc x6 (r)
{
... // this is correct, r may be of any type, even of type ring
}

proc x7 (ring r)
{
... // this is NOT CORRECT
}
```

3.7.3 Help string

Syntax: string_constant;

Purpose: Constitutes the help text of a procedure.

Format:

```
USAGE:    <proc_name>(<parameter list>);   <explanation of parameters>
ASSUME:  <description of assumptions made>
RETURN:   <description of what is returned>
SIDE EFFECTS:  <description of global objects generated or manipulated,
but not returned>
REMARKS: <information on theory and implemented  algorithms,references>
NOTE:    <particularities, limitations, additional details>
KEYWORDS: <semicolon-separated phrases of index keys>
SEE ALSO: <comma-separated names of related procedures/cross references>
EXAMPLE:  example <proc_name>; shows an example
```

NOTE:

- ASSUME, SIDE EFFECTS, KEYWORDS, and SEE ALSO are optional. No help string is required for static procedures.
- EXAMPLE: refers to the example section of the procedure. In a Singular session, the example will be carried out upon entering `example <proc_name>;` if the procedure is loaded from a file by the `LIB` or `load` command (see Section 5.1.77

[LIB], page 208 and see Section 5.2.12 [load], page 293). No example section is allowed if the procedure is defined interactively.
- See Section 3.8.10 [Typesetting of help and info strings], page 64 for help strings in the SINGULAR documentation.
- See the example in Section 3.8.6 [Procedures in a library], page 58 for an illustration.

3.7.4 Names in procedures

- All variables defined inside a procedure are local to the procedure and their names cannot interfere with names in other procedures. Without further action, they are automatically deleted after leaving the procedure.
- To keep local variables and their value after leaving the procedure, they have to be exported (i.e. made global) by a command like `export` or `exportto` (see Section 5.2.6 [export], page 287,see Section 5.2.7 [exportto], page 287, see Section 5.2.10 [importfrom], page 291; see Section 4.15 [package], page 118). To return the value of a local variable, use the `return` command (see Section 5.2.14 [return], page 295).

Example:

```
proc xxx
{
  int k=4;        //defines a local variable k
  int result=k+2;
  export(result);  //defines the global variable "result".
}
xxx();
listvar(all);
↦ // result                         [0]  int 6
```

Note that the variable `result` became a global variable after the execution of `xxx`.

3.7.5 Procedure-specific commands

A few commands should only be used inside a procedure. They either make local objects global ones or return results to the level from where the procedure was called.

See Section 5.2.6 [export], page 287; Section 5.2.7 [exportto], page 287; Section 5.2.14 [return], page 295.

3.8 Libraries

- A library is a collection of SINGULAR procedures in a file.
- To load a library into a SINGULAR session, use the `LIB` or `load` command. Having loaded a library, its procedures can be used like any built-in SINGULAR function, and information on the library is obtained by entering `help libname.lib;`
- See Appendix D [SINGULAR libraries], page 721, for all libraries currently distributed with SINGULAR.
- When writing your own library, it is important to comply with the guidelines described in this section. Otherwise, due to potential parser errors, it may not be possible to load the library.
- Each library consists of a header and a body. The first line of a library must start with a double slash `//`.

- The library header consists of a version string, a category string, an info string, and LIB commands. The strings are mandatory. LIB commands are meant to load the additional libraries used by the library under consideration.
- The library body collects the procedures (declared static or not).
- No line of a library should consist of more than 60 characters.

3.8.1 Libraries in the SINGULAR Documentation

- The typesetting language in which the SINGULAR documentation is written is `texinfo`. The info string of a library included in the SINGULAR distribution will be parsed and automatically translated to the `texinfo` format. The same applies to the help string of each procedure listed in the PROCEDURE: section of the info string.
- Based on various tools, `info`, `dvi`, `p`, and `html` versions of the `texinfo` documentation are generated.
- For texinfo markup elements and other information facilitating optimal typesetting, see Section 3.8.10 [Typesetting of help and info strings], page 64.
- For the convenience of users checking directly the source code, the `texinfo` tools should be used economically. That is, the info and help texts should be well readable verbatim.
- The example of each procedure listed in the PROCEDURE: section of the info string is computed and its output is included in the documentation.

3.8.2 Version string

A version string is part of the header of a library.

Syntax: version = string_constant;

Purpose: Defines the version number of a library. It is displayed when the library is loaded.

Example: `version="version sample.lib 4.0.0.0 Dec_2013 ";`

Note: Syntax: `version<space><filename><space><version><space><date><space>`

3.8.3 Category string

A category string is part of the header of a library.

Syntax: category = string_constant;

Purpose: Defines the category of a library.

Example: category="Algebraic geometry";

Note: Reserved for sorting the libraries into categories.

3.8.4 Info string

Syntax: info = string_constant;

Purpose: Constitutes the help text of a library. Will be displayed in a SINGULAR session upon entering `help libname.lib;` . Will be part of the SINGULAR documentation if the library is distributed with SINGULAR. See Section 3.8.1 [Libraries in the SINGULAR Documentation], page 56.

Format:

```
info="
LIBRARY: <library_name> <one line description of the purpose>
AUTHOR:  <name, and email address of author>
OVERVIEW: <concise, additional information on what is implemented>
REFERENCES: <references for further information>
KEYWORDS: <semicolon-separated phrases of index keys>
SEE ALSO: <comma-separated words of cross references>
PROCEDURES:
  <proc_name_1>();     <one line description of the purpose>
     .
     .
  <proc_name_N>();     <one line description of the purpose>
";
```

NOTE:

- In the documentation, the one line description of the purpose following LIBRARY: will be printed in its own line, starting with the prefix PURPOSE: .
- REFERENCES, KEYWORDS, and SEE ALSO are optional.
- Only non-static procedures should be listed in the PROCEDURES: section. A procedure parameter should be included between the brackets () only if the corresponding one line description of the purpose refers to it. See Section 3.8.6 [Procedures in a library], page 58.
- In the documentation, separate nodes (subsections in printed documents) are created precisely for those procedures of the library appearing n the PROCEDURES: section (that is, for some if not all non-static procedures of the library).

Example:

```
info="
LIBRARY: absfact.lib   Absolute factorization for characteristic 0
AUTHORS: Wolfram Decker,       decker at math.uni-sb.de
         Gregoire Lecerf,      lecerf at math.uvsq.fr
         Gerhard Pfister,      pfister at mathematik.uni-kl.de

OVERVIEW:
A library for computing the absolute factorization of multivariate
polynomials f with coefficients in a field K of characteristic zero.
Using Trager's idea, the implemented algorithm computes an absolutely
irreducible factor by factorizing over some finite extension field L
(which is chosen such that V(f) has a smooth point with coordinates in L).
Then a minimal extension field is determined making use of the
Rothstein-Trager partial fraction decomposition algorithm.

REFERENCES:
G. Cheze, G. Lecerf: Lifting and recombination techniques for absolute
                  factorization. Journal of Complexity, 23(3):380-420, 2007.

KEYWORDS: factorization; absolute factorization.
SEE ALSO: factorize

PROCEDURES:
  absFactorize();        absolute factorization of poly
";
```

To see how this infostring appears in the documentation after typesetting, check Section D.4.1 [absfact_lib], page 741:

3.8.5 LIB commands

LIB commands are part of the header of a library.

Syntax: LIB "lib_1.lib";
...
LIB "lib_r.lib";

Purpose: Loads libraries used by the library under consideration.

Example:

```
LIB "primdec.lib";
LIB "normal.lib";
```

Note: The keyword LIB must be followed by at least one space.

3.8.6 Procedures in a library

Here asre hints and requirements on how procedures contained in a library should be implemented. For more on procedures, see Section 3.7 [Procedures], page 51.

1. Each procedure not meant to be accessible by users should be declared static.
2. The header of each procedure not declared static must comply with the guidelines described in Section 3.7.1 [Procedure definition], page 51 and Section 3.7.3 [Help string], page 54. In particular, it must have a help and example section, and assumptions made should be carefully explained. If the assumptions are checked by the procedure on run-time, errors may be reported using the Section 5.1.29 [ERROR], page 175 function.
3. Names of procedures should not be shorter than 4 characters and should not contain any special characters. In particular, the use of `_` in names of procedures is discourraged. If the name of the procedure is composed of more than one word, each new word should start with a capital letter, all other letters should be lower case (e.g. linearMapKernel).
4. No procedures should be defined within the body of another procedure.
5. A procedure may print out comments, for instance to explain results or to display intermediate computations. This is often helpful when calling the procedure directly, but it may also cause confusions in cases where the procedure is called by another procedure. The SINGULAR solution to this problem makes use of the function `dbprint` (see Section 5.1.16 [dbprint], page 165) and the reserved variables `printlevel` and `voice` (see Section 5.3.6 [printlevel], page 299 and see Section 5.3.11 [voice], page 303). Note that `printlevel` is a predefined, global variable whose value can be changed by the user, while `voice` is an internal variable, representing the nesting level of procedures. Accordingly, the value of Section 5.3.11 [voice], page 303 is 1 on the top level, 2 inside the first procedure, and so on. The default value of `printlevel` is 0, but `printlevel` can be set to any integer value by the user.

Example: If the procedure `Test` below is called directly from the top level, then ‘comment1’ is displayed, but not ‘comment2’. By default, nothing is displayed if `Test` is called from within any other procedure. However, if `printlevel` is set to a value k with k>0, then ‘comment1’ (resp. ‘comment2’) is displayed – provided `Test` is called from another procedure with nesting level at most k (resp. k-1).

The example part of a procedure behaves in this respect like the procedure on top level (the nesting level is 1, that is, the value of `voice` is 2). Therefore, due to the command `printlevel=1;`, ‘comment1’ will be displayed when entering `example Test;`. However, since printlevel is a global variable, it should be reset to its old value at the end of the example part.

The predefined variable `echo` controls whether input lines are echoed or not. Its default is 0, but it can be reset by the user. Input is echoed if `echo>=voice`. At the beginning of the example part, `echo` is set to the value 2. In this way, the input lines of the example will be displayed when entering `example Test;`.

```
proc Test
"USAGE:   ...
         ...
EXAMPLE: example Test; shows an example
"
{   ...
   int p = printlevel - voice + 3;
    ...
   dbprint(p,"comment1");
   dbprint(p-1,"comment2");
   // dbprint prints only if p > 0
    ...
}
example
{ "EXAMPLE:"; echo = 2;
   int p = printlevel;   //store old value of printlevel
   printlevel = 1;       //assign new value to printlevel
    ...
   Test();
   printlevel = p;       //reset printlevel to old value
}
```

Note: SINGULAR functions such as `pause` or `read` allow and require interactive user-input. They are, thus, in particular useful for debugging purposes. If such a command is used inside the procedure of a library to be distributed with SINGULAR, the example section of the procedure has to be written with some care – the procedure should only be called from within the example if the value of `printlevel` is 0. Otherwise, the automatic build process of SINGULAR will not run through since the examples are carried out during the build process. They are, thus, tested against changes in the code.

3.8.7 template_lib

First, we show the source-code of a template library:

```
////////////////////////////////////////////////////////////////
version="version template.lib 4.1.1.0 Dec_2017 "; // $Id: eecdbafd0811d291ba6adc1fc144d110da851fcd $
category="Miscellaneous";
// summary description of the library
info="
LIBRARY:   template.lib  A Template for a Singular Library
AUTHOR:    Olaf Bachmann, email: obachman@mathematik.uni-kl.de

SEE ALSO:  standard_lib, Libraries,
           Typesetting of help and info strings

KEYWORDS: library, template.lib; template.lib; library, info string
```

```
PROCEDURES:
  mdouble(int)           return double of int argument
  mtriple(int)          return three times int argument
  msum([int,..,int])     sum of int arguments
";
////////////////////////////////////////////////////////////////////
proc mdouble(int i)
"USAGE:    mdouble(i); i int
RETURN:   int: i+i
NOTE:     Help string is in pure ASCII.
          This line starts on a new line since previous line is short.
          No new line here.
SEE ALSO: msum, mtriple, Typesetting of help and info strings
KEYWORDS: procedure, ASCII help
EXAMPLE:  example mdouble; shows an example"
{
  return (i + i);
}
example
{ "EXAMPLE:"; echo = 2;
  mdouble(0);
  mdouble(-1);
}
////////////////////////////////////////////////////////////////////
proc mtriple(int i)
"@c we do texinfo here
@table @asis
@item @strong{Usage:}
@code{mtriple(i)}; @code{i} int

@item @strong{Return:}
int: @math{i+i+i}
@item @strong{Note:}
Help is in pure Texinfo.
@*This help string is written in texinfo, which enables you to use,
among others, the @@math command for mathematical typesetting
(for instance, to print @math{\alpha, \beta}).
@*Texinfo also gives more control over the layout, but is, admittingly,
more cumbersome to write.
@end table
@c use @c ref contstuct for references
@cindex procedure, texinfo help
@c ref
@strong{See also:}
@ref{mdouble}, @ref{msum}, @ref{Typesetting of help and info strings}
@c ref
"
{
  return (i + i + i);
}
example
{ "EXAMPLE:"; echo = 2;
```

```
  mtriple(0);
  mtriple(-1);
}
///////////////////////////////////////////////////////////////////
proc msum(list #)
"USAGE:  msum([i_1,..,i_n]); @code{i_1,..,i_n} def
RETURN:  Sum of int arguments
NOTE:    This help string is written in a mixture of ASCII and texinfo.
         @* Use @ref for references (e.g.,  @pxref{mtriple}).
         @* Use @code for typewriter font (e.g., @code{i_1}).
         @* Use @math for simple math mode typesetting (e.g., @math{i_1}).
         @* Warning: Parenthesis like } are not allowed inside @math and @code.
         @* Use @example for indented, preformatted text typesetting in
         typewriter font:
@example
         this  --> that
@end example
        Use @format for preformatted text typesetting in normal font:
@format
         this --> that
@end format
        Use @texinfo for text in pure texinfo:
@texinfo
@expansion{}
@tex
$i_{1,1}$
@end tex

@end texinfo
        Note that
        automatic linebreaking         is still in affect (like in this line).
SEE ALSO: mdouble, mtriple, Typesetting of help and info strings
KEYWORDS: procedure, ASCII/Texinfo help
EXAMPLE: example msum; shows an example"
{
  if (size(#) == 0) { return (0);}
  if (size(#) == 1) { return (#[1]);}
  int i;
  def s = #[1];
  for (i=2; i<=size(#); i++)
  {
    s = s + #[i];
  }
  return (s);
}
example
{ "EXAMPLE:"; echo = 2;
  msum();
  msum(4);
  msum(1,2,3,4);
}
```

Second, we show how the library appears in the documentation after typesetting (with one subsection for each procedure):

Library: template.lib

Purpose: A Template for a Singular Library

Author: Olaf Bachmann, email: obachman@mathematik.uni-kl.de

Procedures: See also: Section 3.8 [Libraries], page 55; Section 3.8.10 [Typesetting of help and info strings], page 64; Section D.1 [standard_lib], page 721.

3.8.7.1 mdouble

Procedure from library `template.lib` (see Section 3.8.7 [template_lib], page 59).

Usage: mdouble(i); i int

Return: int: `i+i`

Note: Help string is in pure ASCII.
This line starts on a new line since previous line is short. No new line here.

Example:

```
LIB "template.lib";
mdouble(0);
↦ 0
mdouble(-1);
↦ -2
```

See also: Section 3.8.10 [Typesetting of help and info strings], page 64; Section 3.8.7.3 [msum], page 62; Section 3.8.7.2 [mtriple], page 62.

3.8.7.2 mtriple

Procedure from library `template.lib` (see Section 3.8.7 [template_lib], page 59).

Usage: `mtriple(i)`; `i` int

Return: int: $i + i + i$

Note: Help is in pure Texinfo.
This help string is written in texinfo, which enables you to use, among others, the `@math` command for mathematical typesetting (for instance, to print α, β).
Texinfo also gives more control over the layout, but is, admittingly, more cumbersome to write.

See also:

Example:

```
LIB "template.lib";
mtriple(0);
↦ 0
mtriple(-1);
↦ -3
```

3.8.7.3 msum

Procedure from library `template.lib` (see Section 3.8.7 [template_lib], page 59).

Usage: msum([i_1,..,i_n]); `i_1,..,i_n` def

Return: Sum of int arguments

Note: This help string is written in a mixture of ASCII and texinfo.
Use @ref for references (e.g., see Section 3.8.7.2 [mtriple], page 62).
Use @code for typewriter font (e.g., `i_1`).
Use @math for simple math mode typesetting (e.g., i_1).
Warning: Parenthesis like } are not allowed inside @math and @code.
Use @example for indented, preformatted text typesetting in typewriter font:

```
this  --> that
```

Use @format for preformatted text typesetting in normal font:

this –> that

Use @texinfo for text in pure texinfo:

$\mapsto i_{1,1}$

Note that
automatic linebreaking is still in affect (like in this line).

Example:

```
LIB "template.lib";
msum();
↦ 0
msum(4);
↦ 4
msum(1,2,3,4);
↦ 10
```

See also: Section 3.8.10 [Typesetting of help and info strings], page 64; Section 3.8.7.1 [mdouble], page 62; Section 3.8.7.2 [mtriple], page 62.

3.8.8 Formal Checker

There is a formal library checker for SINGULAR which can be used online: see `http://www.singular.uni-kl.de/index.php/new-libraries/formal-library-checker.html`.

After uploading your library file, you will receive an output of hints, warnings, and errors which may help you to improve your library.

3.8.9 Documentation Tool

`lib2doc` is a utility to generate the stand-alone documentation for a Singular library in various formats.

The `lib2doc` utility should be used by developers of Singular libraries to check the generation of the documentation of their libraries.

`lib2doc` can be downloaded from
`ftp://www.mathematik.uni-kl.de/pub/Math/Singular/misc/lib2doc.tar.gz`

Important:
To use `lib2doc`, you need to have `perl` (version 5 or higher), `texinfo` (version 3.12 or higher) and `Singular` and `libparse` (version 1-3-4 or higher) installed on your system.

To generate the documentation for a library, follow these steps:

1. Unpack lib2doc.tar.gz

   ```
   gzip -dc  lib2doc.tar.gz | tar -pxf -
   ```

   and

   ```
   cd lib2doc
   ```

2. Edit the beginning of the file `Makefile`, filling in the values for `SINGULAR` and `LIBPARSE`. Check also the values of `PERL` and `LATEX2HTML`.
3. Copy your library to the current directory:
   ```
   cp <path-where-your-lib-is>/mylib.lib .
   ```
4. Now you can run the following commands:

   `make mylib.hlp`
   : Generates the file `mylib.hlp` – the info file for the documentation of `mylib.lib`. This file can be viewed using
     ```
     info -f mylib.hlp
     ```

   `make mylib.dvi`
   : Generates the file `mylib.dvi` – the dvi file for the documentation of `mylib.lib`. This file can be viewed using
     ```
     xdvi mylib.dvi
     ```

   `make mylib.ps`
   : Generates the file `mylib.ps` – the PostScript file for the documentation of `mylib.lib`. This file can be viewed using (for example)
     ```
     ghostview mylib.dvi
     ```

   `make mylib.html`
   : Generates the file `mylib.html` – the HTML file for the documentation of `mylib.lib`. This file can be viewed using (for example)
     ```
     firefox mylib.html
     ```

   `make clean`
   : Deletes all generated files.

   Note that you can safely ignore messages complaining about undefined references.

3.8.10 Typesetting of help and info strings

The info strings of the libraries which are included in the distribution of Singular and the help strings of the corresponding procedures are parsed and automatically converted into the `texinfo` format (the typesetting language in which the documentation of Singular is written).

The illustrative example given in Section 3.8.7 [template_lib], page 59 should provide sufficient information on how this works. For more details, check the following items:

- Users familiar with `texinfo` may write help and info strings directly in the `texinfo` format. The string should, then, start with the `@` sign. In this case, no parsing will be done.
- Help and info strings are typeset within a `@table @asis` environment (which is similar to the `latex description` environment).
- If a line starts with uppercase words up to a colon, then the text up to the colon is taken to be the description-string of an item, and the text following the colon is taken to be the content of the item.
- If the description-string of an item matches

  `SEE ALSO`
  : then the content of the item is assumed to consist of comma-separated words which are valid references to other `texinfo` nodes of the manual (e.g., all procedure and command names are also `texinfo` nodes).

  `KEYWORDS`
  : then the content of the item is assumed to be a semicolon-separated list of phrases which are taken as keys for the index of the manual (the name of a procedure/library is automatically added to the index keys).

- If the description-string of an item in the **info string of a library** matches

`LIBRARY` then the content of the item is assumed to be a one-line description of the library. If this one-line description consists of uppercase characters only, then it is typeset in lowercase characters (otherwise it is left as is).

`PROCEDURES`
then the content of the item is assumed to consist of lines of type

```
<proc_name>();     <one line description of the purpose>
```

Separate `texinfo` nodes (subsections in printed documents) are created precisely for those procedures of the library appearing here (that is, for some if not all non-static procedures of the library).

With respect to the content of an item, the following `texinfo` markup elements are recognized:

`@*` Enforces a line-break.

Example: `old line @* new line`
↦
old line
new line

`@ref{...}` For references to other parts of the SINGULAR manual, use one of the following `@ref{node}` constructs. Here, `node` must be the name of a section of the SIGNULAR manual. In particular, it may be the name of a function, library or procedure in a library.

`@xref{node}`
for a reference to the node `node` at the beginning of a sentence.

`@ref{node}`
for a reference to the node `node` at the end of a sentence.

`@pxref{node}`
for a reference to the node `node` within parentheses.

Example: `@xref{Hurricanes}, for more info.`
↦*Note Hurricanes::, for more info.
↦See Section 3.1 [Hurricanes], page 24, for more info.

`For more information, see @ref{Hurricanes}.`
↦For more information, see *Note Hurricanes::.
↦For more information, see Section 3.1 [Hurricanes], page 24.

`... storms cause flooding (@pxref{Hurricanes}) ...`
↦... storms cause flooding (*Note Hurricanes::) ...
↦... storms cause flooding (see Section 3.1 [Hurricanes], page 24)

`@math{..}` Typeset short mathematical expressions in LaTeX math-mode syntax (short: does not cause expansion over multiple lines).

Example: `@math{\alpha}`
↦
α

Note: The mathematical expressions inside `@math{..}` must not contain the characters `{`,`}`, and `@`.

`@code{..}` Typeset short strings in typewriter font (short: does not cause expansion over multiple lines).

Example: `@code{typewriter font}`
↦
`typewriter font`

Note: The string inside `@code{..}` must not contain the characters `{`,`}`, and `@`.

Typeset pre-formatted text in typewriter font.

```
@example
 ...
@end example
```

Example:

```
before example
@example
in example
notice escape of special characters like @{,@},@@
@end example
after example
```

↦
before example

```
in example
notice escape of special characters like {,},@
```

after example

Note: Inside an `@example` environment, the characters `{`,`}`,`@` have to be escaped by an `@` sign.

Typeset pre-formatted text in normal font.

```
@format
 ...
@end format
```

Example:

```
before format
@format
in format
notice escape of special characters like @{,@},@@
@end format
after format
```

↦
before format

in format
escape of special characters like `{`,`}`,`@`

after format

Note: Inside an `@format` environment, the characters `{`,`}`,`@` have to be escaped by an `@` sign.

Write text in pure `texinfo`.

```
@texinfo
 ...
@end texinfo
```

Example:

```
@texinfo
Among others, within a texinfo environment,
one can use the tex environment to typeset
more complex mathematical items like
@tex
$i_{1,1} $
@tex
@end texinfo
```

↦
Among others, within a texinfo environment, one can use the tex environment to typeset more complex mathematical items like $i_{1,1}$

Furthermore, a line-break is inserted before each line whose previous line is shorter than 60 characters and does not contain any of the above described recognized texinfo markup elements.

3.8.11 Loading a library

Libraries can be loaded with the `LIB` or the `load` command (see Section 5.1.77 [LIB], page 208 and see Section 5.2.12 [load], page 293).

Syntax: `LIB` string_expression `;`
`load` string_expression `;`

Type: none

Purpose: Reads a library from a file. If the given filename does not start with `.` or `/` and if the file cannot be located in the current directory, the `SearchPath` is checked for a directory containing a file with this name.

Note on SearchPath:
The `SearchPath` for a library is constructed at Singular start-up time as follows:

1. the directories contained in the environment variable `SINGULARPATH` are appended.
2. the directories `$BinDir/LIB`, `$RootDir/LIB`, `$RootDir/../LIB`, `$DefaultDir/LIB`, `$DefaultDir/../LIB` are appended, where
   - `$BinDir` is the value of the environment variable `SINGULAR_BIN_DIR`, if set, or, if not set, the directory in which the Singular program resides
   - `$RootDir` is the value of the environment variable `SINGULAR_ROOT_DIR`, if set, or, if not set, `$BinDir/../`.
   - `$DefaultDir` is the value of the environment variable `SINGULAR_DEFAULT_DIR`, if set, or `/usr/local/Singular/` on a Unix platform, `\Singular\` on a Windows 95/98/NT/XP/Vista platform.
3. all directories which do not exist are removed from the `SearchPath`.

For setting environment variables, see Section 5.1.151 [system], page 270, or consult the manual of your shell.

The library `SearchPath` can be examined by starting up Singular with the option `-v`, or by issuing the command `system("--version");`.

Note on standard.lib:
Unless Singular is started with the `--no-stdlib` option, the library `standard.lib` is automatically loaded at start-up time.

Following a `LIB` or `load` command, only the names of the procedures in the library are loaded. The body of a particular procedure is only read upon the first call of the procedure. This minimizes memory consumption by unused procedures. Starting a SINGULAR session with the `-q` or `--quiet` `option` unsets the `option` `loadLib` and inhibits, thus, the monitoring of library loading (see `option`).

All libraries loaded in a SINGULAR session are displayed upon entering `listvar(package);` :

```
option(loadLib);   // show loading of libraries;
                   // standard.lib is loaded
listvar(package);
↦ // Singmathic                    [0]  package Singmathic (C,singmathic.s\
  o)
↦ // Standard                      [0]  package Standard (S,standard.lib)
↦ // Top                           [0]  package Top (T)
                   // the names of the procedures of inout.lib
LIB "inout.lib";   // are now known to Singular
↦ // ** loaded inout.lib (4.1.1.0,Dec_2017)
listvar(package);
↦ // Inout                         [0]  package Inout (S,inout.lib)
↦ // Singmathic                    [0]  package Singmathic (C,singmathic.s\
  o)
↦ // Standard                      [0]  package Standard (S,standard.lib)
↦ // Top                           [0]  package Top (T)
```

See Section 3.1.6 [Command line options], page 19; Section 5.1.77 [LIB], page 208; Section 2.3.3 [Procedures and libraries], page 10; Appendix D [SINGULAR libraries], page 721; Section 4.17 [proc], page 122; Section D.1 [standard_lib], page 721; Section 4.20 [string], page 128; Section 5.1.151 [system], page 270.

3.9 Debugging tools

If SINGULAR does not come back to the prompt while calling a user defined procedure, probably a bracket or a `"` is missing. The easiest way to leave the procedure is to type some brackets or `"` and then ⟨RETURN⟩ .

3.9.1 ASSUME

Syntax: `ASSUME (` int_constant `,` expression `)`

Purpose: Tests the expression for correctness if the int_constant is smaller as a variable `assumeLevel`. If no such variable exist the int expression is compared against 0. It is possible to define an individual `assumeLevel` for each library and/or procedure If the expression is evaluated and not true (i.e. does not evaluate to `int(0)` an error is raised.

Note: `ASSUME shall be used for documentation and debugging,`
production code of a library must never define `assumeLevel`.

Example:

```
ASSUME(0,2==2); // always tested
ASSUME(1,1==2); // not evaluated
int assumeLevel=2;
ASSUME(1,1==2);
↦    ? ASSUME failed:  ASSUME(1,1==2);
```

```
↦    ? error occurred in or before ./examples/ASSUME.sing line 4: ‘  ASSUME\
   (1,1==2);‘
  // setting a different assumeLevel for poly.lib:
  int Poly::assumeLevel=2;
↦ Poly of type ’ANY’. Trying load.
↦ // ** redefining assumeLevel (  int Poly::assumeLevel=2;)
```

3.9.2 Tracing of procedures

Setting the `TRACE` variable to 1 (resp. 3) results in reporting of all procedure entries and exits (resp. together with line numbers). If `TRACE` is set to 4, `Singular` displays each line before its interpretation and waits for the ⟨RETURN⟩ key being pressed. See Section 5.3.9 [TRACE var], page 301.

Example:

```
  proc t1
  {
    int i=2;
    while (i>0)
    { i=i-1; }
  }
  TRACE=3;
  t1();
↦
↦ entering t1 (level 0)
↦ {1}{2}{3}{4}{5}{4}{5}{6}{7}{4}{5}{6}{7}{4}{6}{7}{8}
↦ leaving  t1 (level 0)
```

3.9.3 Source code debugger

The source code debugger (sdb) is an experimental feature, its interface may change in future versions of Singular.
To enable the use of the source code debugger Singular has to be started with the option `-d` or `--sdb` (see Section 3.1.6 [Command line options], page 19).

sdb commands

Each sdb command consists of one character which may be followed by a parameter.

`b` print backtrace of calling stack

`c` continue

`e` edit the current procedure and reload it (current call will be aborted)
only available on UNIX systems

`h,?` display help screen

`n` execute current line, sdb break at next line

`p` <identifier>
display type and value of the variable given by <identifier>

`Q` quit this Singular session

`q` <flags> quit debugger, set debugger flags(0,1,2)
0: continue, disable the debugger
1: continue
2: throw an error, return to toplevel

Syntactical errors in procedures

If Singular was started using the command line option `-d` or `--sdb`, a syntactical error in a procedure will start the source code debugger instead of returning to the top level with an error message. The commands `q 1` and `q 2` are equivalent in this case.

SDB breakpoints in procedures

Up to seven SDB breakpoints can be set. To set a breakpoint at a procedure use `breakpoint`. (See Section 5.2.3 [breakpoint], page 286).
These breakpoints can be cleared with the command `d breakpoint_no` from within the debugger or with `breakpoint(` proc_name `,-1);`.

3.9.4 Break points

A break point can be put into a proc by inserting the command `~`. If `Singular` reaches a break point it asks for lines of commands (line-length must be less than 80 characters) from the user. It returns to normal execution if given an empty line. See Section 5.2.16 [~], page 296.

Example:

```
proc t
{
  int i=2;
  ~;
  return(i+1);
}
t();
↦ -- break point in t --
↦ -- 0: called    from STDIN --
i;               // here local variables of the procedure can be accessed
↦ 2
↦ -- break point in t --

↦ 3
```

3.9.5 Printing of data

The procedure `dbprint` is useful for optional output of data: it takes 2 arguments and prints the second argument, if the first argument is positive; otherwise, it does nothing. See Section 5.1.16 [dbprint], page 165; Section 5.3.11 [voice], page 303.

3.9.6 libparse

`libparse` is a stand-alone program contained in the Singular distribution (at the place where the Singular executable program resides), which cannot be called inside Singular. It is a debugging tool for libraries which performs exactly the same checks as the `load` command in Singular, but generates more output during parsing. `libparse` is useful if an error occurs while loading the library, but the whole block around the line specified seems to be correct. In these situations the real error might have occured hundreds of lines earlier in the library.

Usage:
`libparse [options] singular-library`
Options:

`-d` Debuglevel
: increases the amount of output during parsing, where Debuglevel is an integer between 0 and 4. Default is 0.

`-s`
: turns on reporting about violations of unenforced syntax rules

The following syntax checks are performed in any case:

- counting of pairs of brackets `{,}` , [,] and (,) (number of `{` has to match number of `}`, same for [,] and (,)).
- counting of `"` (number of `"` must be even).
- general library syntax (only LIB, static, proc (with parameters, help, body and example) and comments, i.e // and `/* ... */`, are allowed).

Its output lists all procedures that have been parsed successfully:

```
$ libparse sample.lib
Checking library 'sample.lib'
  Library         function      line,start-eod line,body-eob  line,example-eoe
Version:0.0.0;
g Sample               tab line    9,  149-165   13,  271-298   14,  300-402
l Sample      internal_tab line   24,  450-475   25,  476-496    0,    0-496
```

where the following abbreviations are used:

- g: global procedure (default)
- l: static procedure, i.e., local to the library.

each of the following is the position of the byte in the library.

- start: begin of 'proc'
- eod: end of parameters
- body: start of procedurebody '`{`'
- eob: end of procedurebody '`}`'
- example: position of 'example'
- eoe: end of example '`}`'

Hence in the above example, the first procedure of the library sample.lib is user-accessible and its name is tab. The procedure starts in line 9, at character 149. The head of the procedure ends at character 165, the body starts in line 13 at character 271 and ends at character 298. The example section extends from line 14 character 300 to character 402.

The following example shows the result of a missing close-bracket `}` in line 26 of the library `sample.lib`.

```
LIB "sample.lib";
↦    ? Library sample.lib: ERROR occurred: in line 26, 497.
↦    ? missing close bracket '}' at end of library in line 26.
↦    ? Cannot load library,... aborting.
↦    ? error occurred in STDIN line 1: `LIB "sample.lib";`
```

3.9.7 option(warn)

If this option is set some constructs which **may** lead to bug will result in a warning. While there are legitimate uses for them and they are **not errors** is is worth thinking about it.

change of options during a procedure call: is this side effect intended?

use of `def`: avoids type checking, but usefull if a procedure handles several types at once

ASSUME outside of procedures: while a failed ASSUME aborts the current procedures and return to the top level - what should it do at top level?

See Section 5.1.108 [option], page 230.

3.10 Dynamic loading

In addition to the concept of libraries, it is also possible to dynamically extend the functionality by loading functions written in C/C++ or some other higher programming language. A collection of such functions is called a dynamic module and can be loaded by the command `LIB` or `load`. It is basically handled in the same way as a library: upon loading, a new `package` is created which holds the contents of the dynamic module. General information about the loaded module can be displayed by the command `help package_name`. After loading the dynamic module, its functions can be used exactly like the built-in SINGULAR functions.

To have the full functionality of a built-in function, dynamic modules need to comply with certain requirements on their internal structure. As this would be beyond the scope of the `Singular` manual, a separate, more detailed guide on how to write and use dynamic modules is available.

4 Data types

This chapter explains all data types of SINGULAR in alphabetical order. For every type, there is a description of the declaration syntax as well as information about how to build expressions of certain types.

The term expression list in SINGULAR refers to any comma separated list of expressions.

For the general syntax of a declaration see Section 3.5.1 [General command syntax], page 41.

4.1 cring

Variables of type cring represent the ring of coeffcients (see Section 4.14 [number], page 114)

4.1.1 cring declarations

Syntax: `cring` name `=` cring_expression `;`

Purpose: defines a new coefficient ring resp. field to be used for a ring definition (see Section 4.19 [ring], page 125). Most objects of this type are predefined.

Default: none

Example:

```
  ZZ;
↦ ZZ
  ZZ/3;
↦ ZZ/3
```

4.1.2 cring expressions

A cring expression is:

1. an identifier of type cring
2. a function returning cring
3. an expression involving crings and the arithmetic operations `/`.

Example:

```
  ZZ/3;
↦ ZZ/3
```

See Section 4.19 [ring], page 125.

4.1.3 cring operations

`/` residue class ring

Example:

```
ZZ/101;
↦ ZZ/101
```

4.1.4 cring related functions

`crossprod`
: crooss product of several objects of type cring (see Section 5.1.14 [crossprod], page 165)

`Float`
: several variants of Floating point (inexact) real and complex numbers (see Section 5.1.43 [Float], page 183).

See Section 5.1.14 [crossprod], page 165.

4.2 bigint

Variables of type bigint represent the arbitrary long integers. They can only be contructed from other types (int, number).

4.2.1 bigint declarations

Syntax: `bigint` name = int_expression ;

Purpose: defines a long integer variable

Default: 0

Example:

```
bigint i = 42;
ring r=0,x,dp;
number n=2;
bigint j = i + bigint(n)^50; j;
↦ 1125899906842666
```

4.2.2 bigint expressions

A bigint expression is:

1. an identifier of type bigint
2. a function returning bigint
3. an expression involving bigints and the arithmetic operations `+`, `-`, `*`, `div`, `%` (`mod`), or `^`
4. a type cast to bigint.

Example:

```
// Note: 11*13*17*100*200*2000*503*1111*222222
// returns a machine integer:
11*13*17*100*200*2000*503*1111*222222;
↦ // ** int overflow(*), result may be wrong
↦ // ** int overflow(*), result may be wrong
↦ // ** int overflow(*), result may be wrong
↦ // ** int overflow(*), result may be wrong
↦ -1875651584
// using the type cast number for a greater allowed range
bigint(11)*13*17*100*200*2000*503*1111*222222;
↦ 12075748128684240000000
```

See Section 3.5.5 [Type conversion and casting], page 45; Section 4.6 [int], page 83; Section 4.14 [number], page 114.

4.2.3 bigint operations

`+` addition

`-` negation or subtraction

`*` multiplication

`div` integer division (omitting the remainder >= 0)

`mod`, `%` integer modulo (the remainder of the division `div`)

`^`, `**` exponentiation (exponent must be non-negative)

`<`, `>`, `<=`, `>=`, `==`, `<>`
comparators

Example:

```
bigint(5)*2, bigint(2)^100-10;
↦ 10 1267650600228229401496703205366
bigint(-5) div 2, bigint(-5) mod 2;
↦ -3 1
```

4.2.4 bigint related functions

`gcd` greatest common divisor (see Section 5.1.48 [gcd], page 187)

`memory` memory usage (see Section 5.1.87 [memory], page 217)

See Section 5.1.87 [memory], page 217.

4.3 bigintmat

Big integer matrices are matrices with big integer entries. No basering definition is required to use bigint matrices, for they do not belong to a ring. Bigintmat entries can have any size because of the use of bigint.

4.3.1 bigintmat declarations

Syntax: `bigintmat` name = bigintmat_expression ;
`bigintmat` name [rows] [cols] = bigintmat_expression ;
`bigintmat` name [rows] [cols] = list_of_int_and_bigint expressions ;
rows and cols must be positive int expressions.

Purpose: defines a bigintmat variable.
Given a list of (big) integers, the matrix is filled up with the first row from the left to the right, then the second one and so on. If the (big-)int_list contains less than rows*cols elements, the remaining ones are set to zero; if it contains more elements, only the first rows*cols ones are considered.

Default: empty (1x0 matrix)

Example:

```
  bigintmat bim[4][3]=2, 5, 224553233465, 232444, 434, 0, 0, 4544232222;
  bim;
↦      2,        5,224553233465,
↦ 232444,      434,           0,
↦      0,4544232222,          0,
↦      0,        0,           0
  bim[2, 1];
↦ 232444
```

4.3.2 bigintmat expressions

A bigintmat expression is:

1. an identifier of type bigintmat
2. a function returning bigintmat
3. a bigintmat operation involving (big-)ints and int operations (+, -, *)
4. an expression involving bigintmats and the operations (+, -, *)
5. a type cast to bigintmat (see Section 4.3.3 [bigintmat type cast], page 76)

Example:

```
  bigintmat m1[2][2]=1, 2, 6, 3;
  m1*3;
↦  3,6,
↦ 18,9
  intmat im[3][2] = intmat(m1*3);
  bigintmat m2 = bigintmat(im); // cast intmat im to bigintmat
  m2;
↦  3,6,
↦ 18,9
  m2*m1+m2;
↦ 42,30,
↦ 90,72
  _+4;
↦ 46,34,
↦ 94,76
```

See Section 3.5.5 [Type conversion and casting], page 45; Section 4.3 [bigintmat], page 75.

4.3.3 bigintmat type cast

Syntax: `bigintmat` (expression)

Type: bigintmat

Purpose: Converts expression to a bigintmat, where expression must be of type intmat, or bigintmat.The size (resp. dimension) of the created bigintmat equals the size (resp. dimension) of the expression.

Example:

```
  intmat im[2][1]=2, 3;
  bigintmat(im);
↦ 2,
↦ 3
  bigintmat(_);
```

```
↦ 2,
↦ 3
  bigintmat(intmat(intvec(1,2,3,4), 2, 2)); //casts at first to intmat, then to bigintmat
↦ 1,2,
↦ 3,4
```

See Section 3.5.5 [Type conversion and casting], page 45; Section 4.3 [bigintmat], page 75; Section 4.7.3 [intmat type cast], page 90.

4.3.4 bigintmat operations

`+` addition with intmat, int, or bigint. In case of (big-)int, it is added to every entry of the matrix.

`-` negation or subtraction with intmat, int, or bigint. In case of (big-)int, it is subtracted from every entry of the matrix.

`*` multiplication with intmat, int, or bigint; In case of (big-)int, every entry of the matrix is multiplied by the (big-)int

`<>`, `==` comparators

bigintmat_expression `[` int`,` int `]`
is a bigintmat entry, where the first index indicates the row and the second the column

Example:

```
  bigintmat m[3][4] = 3,3,6,3,5,2,2,7,0,0,45,3;
  m;
↦ 3,3, 6,3,
↦ 5,2, 2,7,
↦ 0,0,45,3
  m[1,3];               // show entry at [row 1, col 3]
↦ 6
  m[1,3] = 10;          // set entry at [row 1, col 3] to 10
  m;
↦ 3,3,10,3,
↦ 5,2, 2,7,
↦ 0,0,45,3
  size(m);         // number of entries
↦ 12
  bigintmat n[2][3] = 2,6,0,4,0,5;
  n * m;
↦ 36,18, 32,48,
↦ 12,12,265,27
  typeof(_);
↦ bigintmat
  -m;
↦ -3,-3,-10,-3,
↦ -5,-2, -2,-7,
↦  0, 0,-45,-3
  bigintmat o;
  o=n-10;
  o;
↦ -8, -4,-10,
```

```
↦ -6,-10, -5
  m*2;           // double each entry of m
↦  6,6,20, 6,
↦ 10,4, 4,14,
↦  0,0,90, 6
  o-2*m;
↦    ? bigintmat/cmatrix not compatible
↦    ? error occurred in or before ./examples/bigintmat_operations.sing lin\
   e 15: `  o-2*m;`
```

4.4 def

Objects may be defined without a specific type: they inherit their type from the first assignment to them. E.g., `ideal i=x,y,z; def j=i^2;` defines the ideal `i^2` with the name `j`.

Note: Unlike other assignments a ring as an untyped object is not a copy but another reference to the same (possibly unnamed) ring. This means that entries in one of these rings appear also in the other ones. The following defines a ring `s` which is just another reference (or name) for the basering `r`. The name `basering` is an alias for the current ring.

```
  ring r=32003,(x,y,z),dp;
  poly f = x;
  def s=basering;
  setring s;
  nameof(basering);
↦ s
  listvar();
↦ // s                                  [0]  *ring
↦ //      f                                  [0]  poly
↦ // r                                  [0]  ring(*)
  poly g = y;
  kill f;
  listvar(r);
↦ // r                                  [0]  ring(*)
↦ // g                                  [0]  poly
  ring t=32003,(u,w),dp;
  def rt=r+t;
  rt;
↦ // coefficients: ZZ/32003
↦ // number of vars : 5
↦ //        block   1 : ordering dp
↦ //                  : names    x y z
↦ //        block   2 : ordering dp
↦ //                  : names    u w
↦ //        block   3 : ordering C
```

This reference to a ring with def is useful if the basering is not local to the procedure (so it cannot be accessed by its name) but one needs a name for it (e.g., for a use with `setring` or `map`). `setring r;` does not work in this case, because `r` may not be local to the procedure.

4.4.1 def declarations

Syntax: `def` name `=` expression `;`

Purpose: defines an object of the same type as the right-hand side.

Default: none

Note: This is useful if the right-hand side may be of variable type as a consequence of a computation (e.g., ideal or module or matrix). It may also be used in procedures to give the basering a name which is local to the procedure.

Example:

```
  def i=2;
  typeof(i);
↦ int
```

See Section 5.1.156 [typeof], page 277.

4.5 ideal

Ideals are represented as lists of polynomials which generate the ideal. Like polynomials they can only be defined or accessed with respect to a basering.

Note: `size` counts only the non-zero generators of an ideal whereas `ncols` counts all generators; see Section 5.1.140 [size], page 260, Section 5.1.101 [ncols], page 227.

4.5.1 ideal declarations

Syntax: `ideal` name = list_of_poly_and_ideal_expressions ;
`ideal` name = ideal_expression ;

Purpose: defines an ideal.

Default: 0

Example:

```
  ring r=0,(x,y,z),dp;
  poly s1 = x2;
  poly s2 = y3;
  poly s3 = z;
  ideal i =  s1, s2-s1, 0,s2*s3, s3^4;
  i;
↦ i[1]=x2
↦ i[2]=y3-x2
↦ i[3]=0
↦ i[4]=y3z
↦ i[5]=z4
  size(i);
↦ 4
  ncols(i);
↦ 5
```

4.5.2 ideal expressions

An ideal expression is:

1. an identifier of type ideal
2. a function returning an ideal
3. a combination of ideal expressions by the arithmetic operations `+` or `*`

4. a power of an ideal expression (operator `^` or `**`)
   Note that the computation of the product `i*i` involves all products of generators of `i` while `i^2` involves only the different ones, and is therefore faster.
5. a type cast to ideal

Example:

```
  ring r=0,(x,y,z),dp;
  ideal m = maxideal(1);
  m;
↦ m[1]=x
↦ m[2]=y
↦ m[3]=z
  poly f = x2;
  poly g = y3;
  ideal i = x*y*z , f-g, g*(x-y) + f^4 ,0, 2x-z2y;
  ideal M = i + maxideal(10);
  timer =0;
  i = M*M;
  timer;
↦ 0
  ncols(i);
↦ 505
  timer =0;
  i = M^2;
  ncols(i);
↦ 505
  timer;
↦ 0
  i[ncols(i)];
↦ x20
  vector v = [x,y-z,x2,y-x,x2yz2-y];
  ideal j = ideal(v);
```

4.5.3 ideal operations

`+` addition (concatenation of the generators and simplification)

`*` multiplication (with ideal, poly, vector, module; simplification in case of multiplication with ideal)

`^` exponentiation (by a non-negative integer)

ideal_expression `[` intvec_expression `]`
: are polynomial generators of the ideal, index 1 gives the first generator.

Note: For simplification of an ideal, see also Section 5.1.139 [simplify], page 258.

Example:

```
  ring r=0,(x,y,z),dp;
  ideal I = 0,x,0,1;
  I;
↦ I[1]=0
↦ I[2]=x
```

```
↦ I[3]=0
↦ I[4]=1
  I + 0;    // simplification
↦ _[1]=1
  ideal J = I,0,x,x-z;;
  J;
↦ J[1]=0
↦ J[2]=x
↦ J[3]=0
↦ J[4]=1
↦ J[5]=0
↦ J[6]=x
↦ J[7]=x-z
  I * J;   //  multiplication with simplification
↦ _[1]=1
  I*x;
↦ _[1]=0
↦ _[2]=x2
↦ _[3]=0
↦ _[4]=x
  vector V = [x,y,z];
  print(V*I);
↦ 0,x2,0,x,
↦ 0,xy,0,y,
↦ 0,xz,0,z
  ideal m = maxideal(1);
  m^2;
↦ _[1]=x2
↦ _[2]=xy
↦ _[3]=xz
↦ _[4]=y2
↦ _[5]=yz
↦ _[6]=z2
  ideal II = I[2..4];
  II;
↦ II[1]=x
↦ II[2]=0
↦ II[3]=1
```

4.5.4 ideal related functions

`char_series`
: irreducible characteristic series (see Section 5.1.6 [char_series], page 159)

`coeffs`
: matrix of coefficients (see Section 5.1.12 [coeffs], page 162)

`contract`
: contraction by an ideal (see Section 5.1.13 [contract], page 164)

`diff`
: partial derivative (see Section 5.1.23 [diff], page 169)

`degree`
: multiplicity, dimension and codimension of the ideal of leading terms (see Section 5.1.19 [degree], page 167)

`dim`
: Krull dimension of basering modulo the ideal of leading terms (see Section 5.1.24 [dim], page 170)

`eliminate`
: elimination of variables (see Section 5.1.27 [eliminate], page 173)

`facstd` factorizing Groebner basis algorithm (see Section 5.1.33 [facstd], page 176)

`factorize`
: ideal of factors of a polynomial (see Section 5.1.35 [factorize], page 178)

`fglm` Groebner basis computation from a Groebner basis w.r.t. a different ordering (see Section 5.1.38 [fglm], page 180)

`finduni` computation of univariate polynomials lying in a zero dimensional ideal (see Section 5.1.42 [finduni], page 182)

`fres` free resolution of a standard basis (see Section 5.1.46 [fres], page 185)

`groebner` Groebner basis computation (a wrapper around `std,stdhilb,stdfglm`,...) (see [groebner], page 721)

`highcorner`
: the smallest monomial not contained in the ideal. The ideal has to be zero-dimensional. (see Section 5.1.53 [highcorner], page 192)

`homog` homogenization with respect to a variable (see Section 5.1.55 [homog], page 194)

`hilb` Hilbert series of a standard basis (see Section 5.1.54 [hilb], page 193)

`indepSet` sets of independent variables of an ideal (see Section 5.1.59 [indepSet], page 196)

`interred` interreduction of an ideal (see Section 5.1.62 [interred], page 199)

`intersect`
: ideal intersection (see Section 5.1.63 [intersect], page 200)

`jacob` ideal of all partial derivatives resp. jacobian matrix (see Section 5.1.64 [jacob], page 200)

`jet` Taylor series up to a given order (see Section 5.1.66 [jet], page 202)

`kbase` vector space basis of basering modulo ideal of leading terms (see Section 5.1.67 [kbase], page 203)

`koszul` Koszul matrix (see Section 5.1.71 [koszul], page 205)

`lead` leading terms of a set of generators (see Section 5.1.73 [lead], page 206)

`lift` lift-matrix (see Section 5.1.78 [lift], page 209)

`liftstd` standard basis and transformation matrix computation (see Section 5.1.79 [liftstd], page 209)

`lres` free resolution for homogeneous ideals (see Section 5.1.81 [lres], page 212)

`maxideal` power of the maximal ideal at 0 (see Section 5.1.86 [maxideal], page 216)

`minbase` minimal generating set of a homogeneous ideal, resp. module, or an ideal, resp. module, in a local ring (see Section 5.1.89 [minbase], page 218)

`minor` set of minors of a matrix (see Section 5.1.90 [minor], page 218)

`modulo` representation of $(h1+h2)/h1 \cong h2/(h1 \cap h2)$ (see Section 5.1.92 [modulo], page 221)

`mres` minimal free resolution of an ideal resp. module w.r.t. a minimal set of generators of the given ideal resp. module (see Section 5.1.96 [mres], page 223)

`mstd` standard basis and minimal generating set of an ideal (see Section 5.1.97 [mstd], page 223) `mult` multiplicity, resp. degree, of the ideal of leading terms (see Section 5.1.98 [mult], page 224)

`ncols` number of columns (see Section 5.1.101 [ncols], page 227)

`nres` a free resolution of an ideal resp. module M which is minimized from the second free module on (see Section 5.1.103 [nres], page 228)

`preimage` preimage under a ring map (see Section 5.1.114 [preimage], page 236)

`qhweight` quasihomogeneous weights of an ideal (see Section 5.1.120 [qhweight], page 242)

`quotient` ideal quotient (see Section 5.1.123 [quotient], page 243)

`reduce` normalform with respect to a standard basis (see Section 5.1.127 [reduce], page 246)

`res` free resolution of an ideal resp. module but not changing the given ideal resp. module (see [res], page 721)

`simplify` simplification of a set of polynomials (see Section 5.1.139 [simplify], page 258)

`size` number of non-zero generators (see Section 5.1.140 [size], page 260)

`slimgb` Groebner basis computation with slim technique (see Section 5.1.141 [slimgb], page 261)

`sortvec` permutation for sorting ideals resp. modules (see Section 5.1.142 [sortvec], page 261)

`sres` free resolution of a standard basis (see Section 5.1.145 [sres], page 264)

`std` standard basis computation (see Section 5.1.147 [std], page 266)

`stdfglm` standard basis computation with fglm technique (see [stdfglm], page 721)

`stdhilb` Hilbert driven standard basis computation (see [stdhilb], page 721)

`subst` substitution of a ring variable (see Section 5.1.150 [subst], page 270)

`syz` computation of the first syzygy module (see Section 5.1.152 [syz], page 275)

`vdim` vector space dimension of basering modulo ideal of leading terms (see Section 5.1.163 [vdim], page 280)

`weight` optimal weights (see Section 5.1.167 [weight], page 283)

4.6 int

Variables of type int represent the machine integers and are, therefore, limited in their range (e.g., the range is between -2147483647 and 2147483647 on 32-bit machines). They are mainly used to count things (dimension, rank, etc.), in loops (see Section 5.2.8 [for], page 290), and to represent boolean values (FALSE is represented by 0, every other value means TRUE, see Section 4.6.5 [boolean expressions], page 87).

Integers consist of a sequence of digits, possibly preceded by a sign. A space is considered as a separator, so it is not allowed between digits. A sequence of digits outside the allowed range is converted to the type `bigint`, see Section 4.2 [bigint], page 74.

4.6.1 int declarations

Syntax: `int` name `=` int_expression `;`

Purpose: defines an integer variable.

Default: 0

Example:

```
  int i = 42;
  int j = i + 3; j;
↦ 45
  i = i * 3 - j; i;
↦ 81
  int k;   // assigning the default value 0 to k
  k;
↦ 0
```

4.6.2 int expressions

An int expression is:

1. a sequence of digits (if the number represented by this sequence is too large to fit into the range of integers it is automatically converted to the type number, if a basering is defined)
2. an identifier of type int
3. a function returning int
4. an expression involving ints and the arithmetic operations `+`, `-`, `*`, `div` (`/`), `%` (`mod`), or `^`
5. a boolean expression
6. a type cast to int

Note: Variables of type int represent the compiler integers and are, therefore, limited in their range (see Section 6.1 [Limitations], page 304). If this range is too small the expression must be converted to the type number over a ring with characteristic 0.

Example:

```
12345678901; // too large
↦ 12345678901
typeof(_);
↦ bigint
ring r=0,x,dp;
12345678901;
↦ 12345678901
typeof(_);
↦ bigint
// Note: 11*13*17*100*200*2000*503*1111*222222
// returns a machine integer:
11*13*17*100*200*2000*503*1111*222222;
↦ // ** int overflow(*), result may be wrong
↦ // ** int overflow(*), result may be wrong
↦ // ** int overflow(*), result may be wrong
↦ // ** int overflow(*), result may be wrong
↦ -1875651584
// using the type cast number for a greater allowed range
number(11)*13*17*100*200*2000*503*1111*222222;
↦ 12075748128684240000000
ring rp=32003,x,dp;
12345678901;
↦ 12345678901
typeof(_);
↦ bigint
```

```
intmat m[2][2] = 1,2,3,4;
m;
↦ 1,2,
↦ 3,4
m[2,2];
↦ 4
typeof(_);
↦ int
det(m);
↦ -2
m[1,1] + m[2,1] == trace(m);
↦ 0
! 0;
↦ 1
1 and 2;
↦ 1
intvec v = 1,2,3;
def d =transpose(v)*v;    // scalarproduct gives an 1x1 intvec
typeof(d);
↦ intvec
int i = d[1];             // access the first (the only) entry in the intvec
ring rr=31,(x,y,z),dp;
poly f = 1;
i = int(f);               // cast to int
// Integers may be converted to constant  polynomials by an assignment,
poly g=37;
// define the constant polynomial g equal to the image of
// the integer 37 in the actual coefficient field, here it equals 6
g;
↦ 6
```

See Section 3.5.5 [Type conversion and casting], page 45; Section 4.14 [number], page 114.

4.6.3 int operations

`++` changes its operand to its successor, is itself no int expression

`--` changes its operand to its predecessor, is itself no int expression

`+` addition

`-` negation or subtraction

`*` multiplication

`div` integer division (omitting the remainder), rounding toward 0

`%`, `mod` integer modulo (the remainder of the division

`^`, `**` exponentiation (exponent must be non-negative)

`<`, `>`, `<=`, `>=`, `==`, `<>`
comparators

Note: An assignment `j=i++;` or `j=i--;` is not allowed, in particular it does not change the value of `j`, see Section 6.1 [Limitations], page 304.

Example:

```
  int i=1;
  int j;
  i++; i;  i--; i;
↦ 2
↦ 1
  // ++ and -- do not return a value as in C, cannot assign
  j = i++;
↦ // ** right side is not a datum, assignment ignored
↦ // ** in line >>  j = i++;<<
  // the value of j is unchanged
  j; i;
↦ 0
↦ 2
  i+2, 2-i, 5^2;
↦ 4 0 25
  5 div 2, 8%3;
↦ 2 2
  -5 div 2, -5 mod 2, -5 % 2;
↦ -2 -1 -1
  1<2, 2<=2;
↦ 1 1
```

4.6.4 int related functions

`char` characteristic of the coefficient field of a ring (see Section 5.1.5 [char], page 159)

`deg` degree of a polynomial resp. vector (see Section 5.1.18 [deg], page 167)

`det` determinant (see Section 5.1.22 [det], page 169)

`dim` Krull dimension of basering modulo ideal of leading terms, resp. dimension of module of leading terms (see Section 5.1.24 [dim], page 170)

`extgcd` Bezout representation of gcd (see Section 5.1.32 [extgcd], page 176)

`find` position of a substring in a string (see Section 5.1.41 [find], page 182)

`gcd` greatest common divisor (see Section 5.1.48 [gcd], page 187)

`koszul` Koszul matrix (see Section 5.1.71 [koszul], page 205)

`memory` memory usage (see Section 5.1.87 [memory], page 217)

`mult` multiplicity of an ideal, resp. module, of leading terms (see Section 5.1.98 [mult], page 224)

`ncols` number of columns (see Section 5.1.101 [ncols], page 227)

`npars` number of ring parameters (see Section 5.1.102 [npars], page 228)

`nrows` number of rows of a matrix, resp. the rank of the free module where the vector or module lives (see Section 5.1.104 [nrows], page 229)

`nvars` number of ring variables (see Section 5.1.106 [nvars], page 230)

`ord` degree of the leading term of a polynomial resp. vector (see Section 5.1.109 [ord], page 235)

`par` n-th parameter of the basering (see Section 5.1.111 [par], page 235) `pardeg` degree of a number considered as a polynomial in the ring parameters (see Section 5.1.112 [pardeg], page 236)

`prime` the next lower prime (see Section 5.1.115 [prime], page 237)

`random` a pseudo random integer between the given limits (see Section 5.1.124 [random], page 244)

`regularity`
regularity of a resolution (see Section 5.1.128 [regularity], page 247)

`rvar` test, if the given expression or string is a ring variable (see Section 5.1.135 [rvar], page 254)

`size` number of elements in an object (see Section 5.1.140 [size], page 260)

`trace` trace of an integer matrix (see Section 5.1.153 [trace], page 276)

`var` n-th ring variable of the basering (see Section 5.1.160 [var], page 279)

`vdim` vector space dimension of basering modulo ideal of leading terms, resp. of freemodule modulo module of leading terms (see Section 5.1.163 [vdim], page 280)

4.6.5 boolean expressions

A boolean expression is an int expression used in a logical context:

An int expression `<>` 0 evaluates to *TRUE* (represented by 1), 0 evaluates to *FALSE* (represented by 0).

The following is the list of available comparisons of objects of the same type.

Note: There are no comparisons for ideals and modules, resolutions and maps.

1. integer comparisons:
```
i == j
i != j    // or     i <> j
i <= j
i >= j
i > j
i < j
```
2. number comparisons:
```
m == n
m != n    // or     m <> n
m < n
m > n
m <= n
m >= n
```
   For numbers from Z/p or from field extensions not all operations are useful:
   - 0 is always the smallest element,
   - in Z/p the representatives in the range -(p-1)/2..(p-1)/2 when p>2 resp. 0 and 1 for p=2 are used for comparisons,
   - in field extensions the last two operations (`>=,<=`) yield always TRUE (1) and the `<` and `>` are equivalent to `!=`.
3. polynomial or vector comparisons:

```
f == g
f != g    // or     f <> g
f <= g    // comparing the leading term w.r.t. the monomial order
f <  g
f >= g
f >  g
```

4. intmat or matrix comparisons:

```
v == w
v != w    // or     v <> w
```

5. intvec or string comparisons:

```
f == g
f != g    // or     f <> g
f <= g    // comparing lexicographically
f >= g    // w.r.t. the order specified by ASCII
f >  g
f <  g
```

6. boolean expressions combined by boolean operations (`and`, `or`, `not`)

Note: All arguments of a logical expression are first evaluated and then the value of the logical expression is determined. For example, the logical expression `(a || b)` is evaluated by first evaluating `a` *and* `b`, even though the value of `b` has no influence on the value of `(a || b)`, if `a` evaluates to true.

Note that this evaluation is different from the left-to-right, conditional evaluation of logical expressions (as found in most programming languages). For example, in these other languages, the value of `(1 || b)` is determined without ever evaluating `b`.

See Section 6.3 [Major differences to the C programming language], page 304.

4.6.6 boolean operations

`and` logical `and`, may also be written as `&&`

`or` logical `or`, may also be written as `||`

`not` logical `not`, may also be written as `!`

The precedence of the boolean operations is:

1. parentheses
2. comparisons
3. not
4. and
5. or

Example:

```
  (1>2) and 3;
↦ 0
  1 > 2 and 3;
↦ 0
  ! 0 or 1;
↦ 1
  !(0 or 1);
↦ 0
```

4.7 intmat

Integer matrices are matrices with integer entries. For the range of integers see Section 6.1 [Limitations], page 304. Integer matrices do not belong to a ring, they may be defined without a basering being defined. An intmat can be multiplied by and added to an int; in this case the int is converted into an intmat of the right size with the integer on the diagonal. The integer `1`, for example, is converted into the unit matrix.

4.7.1 intmat declarations

Syntax: `intmat` name = intmat_expression ;
`intmat` name [rows] [cols] = intmat_expression ;
`intmat` name [rows] [cols] = list_of_int_and_intvec_and_intmat_expressions ;
rows and cols must be positive int expressions.

Purpose: defines an intmat variable.
Given a list of integers, the matrix is filled up with the first row from the left to the right, then the second row and so on. If the int_list contains less than rows*cols elements, the matrix is filled up with zeros; if it contains more elements, only the first rows*cols elements are used.

Default: 0 (1 x 1 matrix)

Example:

```
  intmat im[3][5]=1,3,5,7,8,9,10,11,12,13;
  im;
↦ 1,3,5,7,8,
↦ 9,10,11,12,13,
↦ 0,0,0,0,0
  im[3,2];
↦ 0
  intmat m[2][3] = im[1..2,3..5];  // defines a submatrix
  m;
↦ 5,7,8,
↦ 11,12,13
```

4.7.2 intmat expressions

An intmat expression is:

1. an identifier of type intmat
2. a function returning intmat
3. an intmat operation involving ints and int operations (`+`, `-`, `*`, `div`, `%`)
4. an expression involving intmats and the operations (`+`, `-`, `*`)
5. a type cast to intmat (see Section 4.7.3 [intmat type cast], page 90)

Example:

```
  intmat Idm[2][2];
  Idm +1;           // add the unit intmat
↦ 1,0,
↦ 0,1
  intmat m1[3][2] = _,1,-2;  // take entries from the last result
```

```
  m1;
↦ 1,0,
↦ 0,1,
↦ 1,-2
  intmat m2[2][3]=1,0,2,4,5,1;
  transpose(m2);
↦ 1,4,
↦ 0,5,
↦ 2,1
  intvec v1=1,2,4;
  intvec v2=5,7,8;
  m1=v1,v2;         // fill m1 with v1 and v2
  m1;
↦ 1,2,
↦ 4,5,
↦ 7,8
  trace(m1*m2);
↦ 56
```

See Section 3.5.5 [Type conversion and casting], page 45; Section 4.14 [number], page 114.

4.7.3 intmat type cast

Syntax: `intmat` (expression)
`intmat` (expression, int_n, int_m)

Type: intmat

Purpose: Converts expression to an intmat, where expression must be of type intvec, intmat, or bigintmat. If int_n and int_m are supplied, then they specify the dimension of the intmat. Otherwise, the size (resp. dimensions) of the intmat are determined by the size (resp. dimensions) of the expression. If expression is a bigintmat containing an entry larger the the limit of int, it is set to 0 in the returning intmat.

Example:

```
  intmat(intvec(1));
↦ 1
  intmat(intvec(1), 1, 2);
↦ 1,0
  intmat(intvec(1,2,3,4), 2, 2);
↦ 1,2,
↦ 3,4
  intmat(_, 2, 3);
↦ 1,2,3,
↦ 4,0,0
  intmat(_, 2, 1);
↦ 1,2
  bigintmat bim[2][3]=34, 64, 345553234, 35553, 6434, 6563335675;
  intmat(bim);
↦ 34,64,345553234,
↦ 35553,6434,-2026598917
```

See Section 3.5.5 [Type conversion and casting], page 45; Section 4.7 [intmat], page 89; Section 4.12.3 [matrix type cast], page 108.

4.7.4 intmat operations

`+` addition with intmat or int; the int is converted into a diagonal intmat

`-` negation or subtraction with intmat or int; the int is converted into a diagonal intmat

`*` multiplication with intmat, intvec, or int; the int is converted into a diagonal intmat

`div,/` division of entries in the integers (omitting the remainder)

`%, mod` entries modulo int (remainder of the division)

`<>, ==` comparators

intmat_expression `[` intvec_expression`,` intvec_expression `]`
is an intmat entry, where the first index indicates the row and the second the column

Example:

```
  intmat m[2][4] = 1,0,2,4,0,1,-1,0,3,2,1,-2;
  m;
↦ 1,0,2,4,
↦ 0,1,-1,0
  m[2,3];          // entry at row 2, col 3
↦ -1
  size(m);         // number of entries
↦ 8
  intvec v = 1,0,-1,2;
  m * v;
↦ 7,1
  typeof(_);
↦ intvec
  intmat m1[4][3] = 0,1,2,3,v,1;
  intmat m2 = m * m1;
  m2;             //  2 x 3 intmat
↦ -2,5,4,
↦ 4,-1,-1
  m2*10;           // multiply each entry of m with 10;
↦ -20,50,40,
↦ 40,-10,-10
  -m2;
↦ 2,-5,-4,
↦ -4,1,1
  m2 % 2;
↦ 0,1,0,
↦ 0,1,1
  m2 div 2;
↦ -1,2,2,
↦ 2,-1,-1
  m2[2,1];          // entry at row 2, col 1
↦ 4
  m1[2..3,2..3];   // submatrix
↦ 1 0 2 1
  m2[nrows(m2),ncols(m2)];      // the last entry of intmat m2
↦ -1
```

4.7.5 intmat related functions

`betti` Betti numbers of a free resolution (see Section 5.1.4 [betti], page 157)

`det` determinant (see Section 5.1.22 [det], page 169)

`ncols` number of cols (see Section 5.1.101 [ncols], page 227)

`nrows` number of rows (see Section 5.1.104 [nrows], page 229)

`random` pseudo random intmat (see Section 5.1.124 [random], page 244)

`size` total number of entries (see Section 5.1.140 [size], page 260)

`transpose`
transpose of an intmat (see Section 5.1.154 [transpose], page 276)

`trace` trace of an intmat (see Section 5.1.153 [trace], page 276)

4.8 intvec

Variables of type intvec are lists of integers. For the range of integers see Section 6.1 [Limitations], page 304. They may be used for simulating sets of integers (and other sets if the intvec is used as an index set for other objects). Addition and subtraction of an intvec with an int or an intvec is done element-wise.

4.8.1 intvec declarations

Syntax: `intvec` name = intvec_expression `;`
`intvec` name = list_of_int_and_intvec_expressions `;`

Purpose: defines an intvec variable.
An intvec consists of an ordered list of integers.

Default: 0

Example:

```
  intvec iv=1,3,5,7,8;
  iv;
↦ 1,3,5,7,8
  iv[4];
↦ 7
  iv[3..size (iv)];
↦ 5 7 8
```

4.8.2 intvec expressions

An intvec expression is:

1. a range: int expression `..` int expression
2. a repeated entry: int expression `:` positive int expression
(`a:b` generates an `intvec` of length `b>0` with identical entries `a`)
3. a function returning intvec
4. an expression involving intvec operations with int (`+`, `-`, `*`, `/`, `%`)
5. an expression of intvecs involving intvec operations (`+`, `-`)
6. an expression involving an intvec operation with intmat (`*`)

7. a type cast to intvec

Example:

```
  intvec v=-1,2;
  intvec w=v,v;         // concatenation
  w;
↦ -1,2,-1,2
  w=2:3;                // repetition
  w;
↦ 2,2,2
  int k = 3;
  v = 7:k;
  v;
↦ 7,7,7
  v=-1,2;
  w=-2..2,v,1;
  w;
↦ -2,-1,0,1,2,-1,2,1
  intmat m[3][2] = 0,1,2,-2,3,1;
  m*v;
↦ 2,-6,-1
  typeof(_);
↦ intvec
  v = intvec(m);
  v;
↦ 0,1,2,-2,3,1
  ring r;
  poly f = x2z + 2xy-z;
  f;
↦ x2z+2xy-z
  v = leadexp(f);
  v;
↦ 2,0,1
```

4.8.3 intvec operations

`+` addition with intvec or int (component-wise)

`-` negation or subtraction with intvec or int (component-wise)

`*` multiplication with int (component-wise)

`/`, `div` division by int (component-wise)

`%`, `mod` modulo (component-wise)

`<>`, `==`, `<=`, `>=`, `>`, `<`
: comparison (done lexicographically, different length will be filled with 0 at th right)

intvec_expression `[` int_expression `]`
: is an element of the intvec; the first element has index one.

Example:

```
  intvec iv =  1,3,5,7,8;
  iv+1;               // add 1 to each entry
↦ 2,4,6,8,9
  iv*2;
↦ 2,6,10,14,16
  iv;
↦ 1,3,5,7,8
  iv-10;
↦ -9,-7,-5,-3,-2
  iv=iv,0;
  iv;
↦ 1,3,5,7,8,0
  iv div 2;
↦ 0,1,2,3,4,0
  iv+iv;              // component-wise addition
↦ 2,6,10,14,16,0
  iv[size(iv)-1];     // last-1 entry
↦ 8
  intvec iw=2,3,4,0;
  iv==iw;             // lexicographic comparison
↦ 0
  iv < iw;
↦ 1
  iv != iw;
↦ 1
  iv[2];
↦ 3
  iw = 4,1,2;
  iv[iw];
↦ 7 1 3
```

4.8.4 intvec related functions

`hilb` Hilbert series as intvec (see Section 5.1.54 [hilb], page 193)

`indepSet` sets of independent variables of an ideal (see Section 5.1.59 [indepSet], page 196)

`leadexp` the exponent vector of the leading monomial (see Section 5.1.75 [leadexp], page 207)

`monomial` the power product corresponding to the exponent vector (see Section 5.1.94 [monomial], page 222)

`nrows` number of rows (see Section 5.1.104 [nrows], page 229)

`qhweight` quasihomogeneous weights (see Section 5.1.120 [qhweight], page 242)

`size` length of the intvec (see Section 5.1.140 [size], page 260)

`sortvec` permutation for sorting ideals/modules (see Section 5.1.142 [sortvec], page 261)

`transpose`
transpose of an intvec, returns an intmat (see Section 5.1.154 [transpose], page 276)

`weight` weights for the weighted ecart method (see Section 5.1.167 [weight], page 283)

4.9 link

Links are the communication channels of SINGULAR, i.e., something SINGULAR can write to and/or read from. Currently, SINGULAR supports four different link types:

- ASCII links (see Section 4.9.4 [ASCII links], page 96)
- ssi links (see Section 4.9.5 [Ssi links], page 97)
- pipe links (see Section 4.9.6 [Pipe links], page 100)
- DBM links (see Section 4.9.7 [DBM links], page 100)

4.9.1 link declarations

Syntax: `link` name = string_expression ;

Purpose: defines a new communication link.

Default: none

Example:

```
  link l=":w example.txt";
  int i=22;          // cf. ASCII links for explanation
  string s="An int follows:";
  write(l,s,i);
  l;
↦ // type : ASCII
↦ // mode : w
↦ // name : example.txt
↦ // open : yes
↦ // read : not ready
↦ // write: ready
  close(l);          //
  read(l);
↦ An int follows:
↦ 22
↦
  close(l);
```

4.9.2 link expressions

A link expression is:

1. an identifier of type link
2. a string describing the link

A link is described by a string which consists of two parts: a property string followed by a name string. The property string describes the type of the link (`ASCII`, `ssi` or `DBM`) and the mode of the link (e.g., open for read, write or append). The name string describes the filename of the link, resp. a network connection for ssi links.

For a detailed format description of the link describing string see:

- for ASCII links: Section 4.9.4 [ASCII links], page 96
- ssi links (see Section 4.9.5 [Ssi links], page 97)
- pipe links (see Section 4.9.6 [Pipe links], page 100)
- for DBM links: Section 4.9.7 [DBM links], page 100

4.9.3 link related functions

`close` closes a link (see Section 5.1.10 [close], page 161)

`dump` generates a dump of all variables and their values (see Section 5.1.26 [dump], page 172)

`getdump` reads a dump (see Section 5.1.50 [getdump], page 188)

`open` opens a link (see Section 5.1.107 [open], page 230)

`read` reads from a link (see Section 5.1.126 [read], page 245)

`status` gets the status of a link (see Section 5.1.146 [status], page 265)

`write` writes to a link (see Section 5.1.169 [write], page 283)

`kill` closes and kills a link (see Section 5.1.69 [kill], page 204)

`waitall` wait till all links of a list of links become ready (only ssi:tcp links) (see Section 5.1.164 [waitall], page 281)

`waitfirst`
wait till at least one link of a list of links become ready (only ssi:tcp links) (see Section 5.1.165 [waitfirst], page 282)

4.9.4 ASCII links

Via ASCII links data that can be converted to a string can be written into files for storage or communication with other programs. The data is written in plain ASCII format. The output format of polynomials is done w.r.t. the value of the global variable `short` (see Section 5.3.7 [short], page 299). Reading from an ASCII link returns a string — conversion into other data is up to the user. This can be done, for example, using the command `execute` (see Section 5.1.31 [execute], page 175).

The ASCII link describing string has to be one of the following:

1. `"ASCII: " +` filename
   the mode (read or append) is set by the first `read` or `write` command.
2. `"ASCII:r " +` filename
   opens the file for reading.
3. `"ASCII:w " +` filename
   opens the file for overwriting.
4. `"ASCII:a " +` filename
   opens the file for appending.

There are the following default values:

- the type `ASCII` may be omitted since ASCII links are the default links.
- if non of `r`, `w`, or `a` is specified, the mode of the link is set by the first `read` or `write` command on the link. If the first command is `write`, the mode is set to `a` (append mode).
- if the filename is omitted, `read` reads from stdin and `write` writes to stdout.

Using these default rules, the string `":r temp"` describes a link which is equivalent to the link `"ASCII:r temp"`: an ASCII link to the file `temp` which is opened for reading. The string `"temp"` describes an ASCII link to the file `temp`, where the mode is set by the first `read` or `write` command. See also the example below.

Note that the filename may contain a path. On Microsoft Windows (resp. MS-DOS) platforms, names of a drive can precede the filename, but must be started with a `//` (as in `//c/temp/ex`. An

ASCII link can be used either for reading or for writing, but not for both at the same time. A `close` command must be used before a change of I/O direction. Types without a conversion to `string` cannot be written.

Example:

```
  ring r=32003,(x,y,z),dp;
  link l=":w example.txt";     // type is ASCII, mode is overwrite
  l;
↦ // type : ASCII
↦ // mode : w
↦ // name : example.txt
↦ // open : no
↦ // read : not ready
↦ // write: not ready
  status(l, "open", "yes");    // link is not yet opened
↦ 0
  ideal i=x2,y2,z2;
  write (l,1,";",2,";","ideal i=",i,";");
  status(l, "open", "yes");    // now link is open
↦ 1
  status(l, "mode");           // for writing
↦ w
  close(l);                    // link is closed
  write("example.txt","int j=5;");// data is appended to file
  read("example.txt");         // data is returned as string
↦ 1
↦ ;
↦ 2
↦ ;
↦ ideal i=
↦ x2,y2,z2;
↦ int j=5;
↦ 
  execute(read(l));            // read string is executed
↦ 1
↦ 2
↦ // ** redefining i (ideal i=)
  close(l);                    // link is closed
```

4.9.5 Ssi links

Ssi (simple singular interface) links give the possibility to store and communicate data betweenm Singular processes: Read and write access is very fast compared to ASCII links. Ssi links can be established using files or using TCP sockets. For ring-dependent data, a ring description is written together with the data. Reading from an Ssi link returns an expression (not a string) which was evaluated after the read operation. If the expression read from an Ssi link is not from the same ring as the current ring, then a `read` changes the current ring.

Currently under development - not everything is implemtented.

4.9.5.1 Ssi file links

Ssi file links provide the possibility to store data in a file using the ssi format. For storing large amounts of data, ssi file links should be used instead of ASCII links. Unlike ASCII links, data read from ssi file links is returned as expressions one at a time.

The ssi file link describing string has to be one of the following:

1. `"ssi:r " +` filename
   opens the file for reading.
2. `"ssi:w " +` filename
   opens the file for overwriting.
3. `"ssi:a " +` filename
   opens the file for appending.

Note that the filename may contain a path. An ssi file link can be used either for reading or for writing, but not for both at the same time. A `close` command must be used before a change of I/O direction.

Example:

```
  ring r;
  link l="ssi:w example.ssi"; // type=ssi, mode=overwrite
  l;
↦ // type : ssi
↦ // mode : w
↦ // name : example.ssi
↦ // open : no
↦ // read : not open
↦ // write: not open
  ideal i=x2,y2,z2;
  write (l,1, i, "hello world");// write three expressions
  write(l,4);                   // append one more expression
  close(l);                     // link is closed
  // open the file for reading now
  read(l);                      // only first expression is read
↦ 1
  kill r;                       // no basering active now
  def i = read(l);              // second expression
  // notice that current ring was set, the name was assigned
  // automatically
  listvar(ring);
↦ // ssiRing0                        [0]  *ring
↦ // flint_poly_Q                    [0]  cring
↦ // ZZ                              [0]  cring
↦ // QQ                              [0]  cring
  def s = read(l);              // third expression
  listvar();
↦ // s                               [0]  string hello world
↦ // ssiRing0                        [0]  *ring
↦ //      i                               [0]  ideal, 3 generator(s)
↦ // l                               [0]  link
  close(l);                     // link is closed
```

4.9.5.2 Ssi tcp links

Ssi tcp links give the possibility to exchange data between two processes which may run on the same or on different computers. Ssi tcp links can be opened in four different modes:

`tcp` Singular acts as a server.

`connect` Singular acts as a client.

`tcp <host>:<program>`
Singular acts as a client, launching an application as server. This reuires `ssh`/`sshd` to be installed an the computers (and preferably an automatic login via `.ssh/authorized_keys`).

`fork` Singular acts as a client, forking another Singular as server.

The Ssi tcp link describing string has to be

- tcp mode:
  1. `"ssi:tcp"`

  Singular becomes a server and waits at the first free port (>1024) for a connect call.
- connect mode:
  2. `"ssi:connect " +` host:port

  Singular becomes a client and connects to a server waiting at the host and port.
- launch mode:
  4. `"ssi:tcp" +` host:application

  Singular becomes a client and starts (launches) the application using ssh on a (possibly) different host which then acts as a server.
- fork mode:
  8. `"ssi:fork"`

  Singular becomes a client and forks another Singular on the same host which acts as a server.

To open an ssi tcp link in launch mode, the application to launch must either be given with an absolute pathname, or must be in a directory contained in the search path. The launched application acts as a server, whereas the Singular that actually opened the link acts as a client. The client "listens" at the some free port until the server application does a connect call.

If the ssi tcp link is opened in fork mode a child of the current Singular is forked. All variables and their values are inherited by the child. The child acts as a server whereas the Singular that actually opened the link acts as a client.

To arrange the evaluation of an expression by a server, the expression must be quoted using the command `quote` (see Section 5.1.122 [quote], page 243), so that a local evaluation is prevented. Otherwise, the expression is evaluated first, and the result of the evaluation is written, instead of the expression which is to be evaluated.

If Singular is in server mode, the value of the variable `link_ll` is the ssi link connecting to the client and Singular is in an infinite read-eval-write loop until the connection is closed from the client side (by closing its connecting link). Reading and writing is done to the link `link_ll`: After an expression is read, it is evaluated and the result of the evaluation is written back. That is, for each expression which was written to the server, there is exactly one expression written back. This might be an "empty" expression, if the evaluation on the server side does not return a value.

Ssi tcp links should explicitly be opened before being used. Ssi tcp links are bidirectional, i.e. can be used for both, writing and reading. Reading from an ssi tcp link blocks until data was written to that link. The `status` command can be used to check whether there is data to read.

Example:

```
  int i=7;
  link l = "ssi:fork";      // fork link declaration
  open(l); l;
↦ // type : ssi
↦ // mode : fork
↦ // name :
↦ // open : yes
↦ // read : not ready
↦ // write: ready

  write(l,quote(i)); // Child inherited vars and their values
  read(l);
↦ 7
  close(l);          // shut down forked child
```

4.9.6 Pipe links

Pipe links provide acces to stdin and stdout of any program. Pipe links are bidirectional. **Syntax:** `"|: "` + string_for_system

The string_for system will be passed to `system` after conneting the input and output to the corresponding stdout and stdin.

Example:

```
  link l="|: date";
  open(l); l;
↦ // type : pipe
↦ // mode :
↦ // name : date
↦ // open : yes
↦ // read : not ready
↦ // write: ready
  read(l);
↦ Di 13. Feb 13:19:48  2018
  l;
↦ // type : pipe
↦ // mode :
↦ // name : date
↦ // open : yes
↦ // read : not ready
↦ // write: ready
  close(l);
```

4.9.7 DBM links

DBM links provide access to data stored in a data base. Each entry in the data base consists of a (key_string, value_string) pair. Such a pair can be inserted with the command `write(`link`,` key_string`,` value_string`)`. By calling `write(`link`,` key_string`)`, the entry with key key_string is deleted from the data base. The value of an entry is returned by the command `read`(link, key_string). With only one argument, `read`(link) returns the next key in the data base. Using this feature a data base can be scanned in order to access all entries of the data base.

If a data base with name `name` is opened for writing for the first time, two files (`name.pag` and `name.dir`), which contain the data base, are automatically created.

The DBM link describing string has to be one of the following:

1. `"DBM: " +` name
   opens the data base for reading (default mode).
2. `"DBM:r " +` name
   opens the data base for reading.
3. `"DBM:rw " +` name
   opens the data base for reading and writing.

Note that `name` must be given without the suffix `.pag` or `.dir`. The name may contain an (absolute) path.

Example:

```
  link l="DBM:rw example";
  write(l,"1","abc");
  write(l,"3","XYZ");
  write(l,"2","ABC");
  l;
↦ // type : DBM
↦ // mode : rw
↦ // name : example
↦ // open : yes
↦ // read : ready
↦ // write: ready
  close(l);
  // read all keys (till empty string):
  read(l);
↦ 1
  read(l);
↦ 3
  read(l);
↦ 2
  read(l);
↦
  // read data corresponding to key "1"
  read(l,"1");
↦ abc
  // read all data:
  read(l,read(l));
↦ abc
  read(l,read(l));
↦ XYZ
  read(l,read(l));
↦ ABC
  // close
  close(l);
```

4.10 list

Lists are arrays whose elements can be of different types (including ring). If one element belongs to a ring the whole list belongs to that ring. This applies also to the special list `#`. The expression `list()` is the empty list.

Note that a list stores the objects itself and not the names. Hence, if `L` is a list, `L[1]` for example has no name. A name, say `R`, can be created for `L[1]` by `def R=L[1];`. To store also the name of an object, say `r`, it can be added to the list with `nameof(r);`. Rings may be objects of a list.

Note: Unlike other assignments a ring as an element of a list is not a copy but another reference to the same ring.

4.10.1 list declarations

Syntax: `list` name = expression_list;
`list` name = list_expression;

Purpose: defines a list (of objects of possibly different types).

Default: empty list

Example:

```
  list l=1,"str";
  l[1];
↦ 1
  l[2];
↦ str
  ring r;
  listvar(r);
↦ // r                              [0]  *ring
  ideal i = x^2, y^2 + z^3;
  l[3] = i;
  l;
↦ [1]:
↦    1
↦ [2]:
↦    str
↦ [3]:
↦    _[1]=x2
↦    _[2]=z3+y2
  listvar(r);     // the list l belongs now to the ring r
↦ // r                              [0]  *ring
↦ // l                              [0]  list, size: 3
↦ // i                              [0]  ideal, 2 generator(s)
```

4.10.2 list expressions

A list expression is:

1. the empty list `list()`
2. an identifier of type list
3. a function returning list
4. list expressions combined by the arithmetic operation `+`
5. a type cast to list

See Section 3.5.5 [Type conversion and casting], page 45.

Example:

```
  list l = "hello",1;
  l;
↦ [1]:
↦    hello
↦ [2]:
↦    1
  l = list();
  l;
↦ empty list
  ring r =0,x,dp;
  factorize((x+1)^2);
↦ [1]:
↦    _[1]=1
↦    _[2]=x+1
↦ [2]:
↦    1,2
  list(1,2,3);
↦ [1]:
↦    1
↦ [2]:
↦    2
↦ [3]:
↦    3
```

4.10.3 list operations

`+` concatenation

`delete` deletes one element from list, returns new list

`insert` inserts or appends a new element to list, returns a new list

list_expression `[` int_expression `]`
is a list entry; the index 1 gives the first element.

Example:

```
  list l1 = 1,"hello",list(-1,1);
  list l2 = list(1,5,7);
  l1 + l2;             // a new list
↦ [1]:
↦    1
↦ [2]:
↦    hello
↦ [3]:
↦    [1]:
↦       -1
↦    [2]:
↦       1
↦ [4]:
↦    1
```

```
↦ [5]:
↦    5
↦ [6]:
↦    7
  l2 = delete(l2, 2);  // delete 2nd entry
  l2;
↦ [1]:
↦    1
↦ [2]:
↦    7
```

4.10.4 list related functions

`bareiss` returns a list of a matrix (lower triangular) and of an intvec (permutations of columns, see Section 5.1.3 [bareiss], page 156)

`betti` Betti numbers of a resolution (see Section 5.1.4 [betti], page 157)

`delete` deletion of an element from a list (see Section 5.1.20 [delete], page 168)

`facstd` factorizing Groebner basis algorithm (see Section 5.1.33 [facstd], page 176)

`factorize`
list of factors of a polynomial (see Section 5.1.35 [factorize], page 178)

`insert` insertion of a new element into a list (see Section 5.1.60 [insert], page 197)

`minres` minimization of a free resolution (see Section 5.1.91 [minres], page 220)

`names` list of all user-defined variable names (see Section 5.1.100 [names], page 225)

`size` number of entries (see Section 5.1.140 [size], page 260)

`conversion from resolution`
(see Section 4.18 [resolution], page 124)

4.11 map

Maps are ring maps from a preimage ring into the basering.

Note:

- The target of a map is **ALWAYS** the actual basering
- The preimage ring has to be stored "by its name", that means, maps can only be used in such contexts, where the name of the preimage ring can be resolved (this has to be considered in subprocedures). See also Section 6.5 [Identifier resolution], page 310, Section 3.7.4 [Names in procedures], page 55.

Maps between rings with different coefficient fields are possible and listed below.

Canonically realized are

- $Q \to Q(a, \ldots)$ (Q : the rational numbers)
- $Q \to R$ (R : the real numbers)
- $Q \to C$ (C : the complex numbers)
- $Z/p \to (Z/p)(a, \ldots)$ (Z : the integers)
- $Z/p \to GF(p^n)$ (GF : the Galois field)
- $Z/p \to R$

- $R \to C$

Possible are furthermore

- $Z/p \to Q, \quad [i]_p \mapsto i \in [-p/2,\, p/2] \subseteq Z$
- $Z/p \to Z/p', \quad [i]_p \mapsto i \in [-p/2,\, p/2] \subseteq Z,\ i \mapsto [i]_{p'} \in Z/p'$
- $C \to R,$ by taking the real part

Finally, in SINGULAR we allow the mapping from rings with coefficient field Q to rings whose ground fields have finite characteristic:

- $Q \to Z/p$
- $Q \to (Z/p)(a, \ldots)$

In these cases the denominator and the numerator of a number are mapped separately by the usual map from Z to Z/p, and the image of the number is built again afterwards by division. It is thus not allowed to map numbers whose denominator is divisible by the characteristic of the target ground field, or objects containing such numbers. We, therefore, strongly recommend using such maps only to map objects with integer coefficients.

4.11.1 map declarations

Syntax: `map` name = preimage_ring_name , ideal_expression ;
`map` name = preimage_ring_name , list_of_poly_and_ideal_expressions ;
`map` name = map_expression ;

Purpose: defines a ring map from preimage_ring to basering.
Maps the variables of the preimage ring to the generators of the ideal. If the ideal contains less elements than variables in the preimage_ring the remaining variables are mapped to 0, if the ideal contains more elements these are ignored. The image ring is always the current basering. For the mapping of coefficients from different fields see Section 4.11 [map], page 104.

Default: none

Note: There are standard mappings for maps which are close to the identity map: `fetch` and `imap`.
The name of a map serves as the function which maps objects from the preimage_ring into the basering. These objects must be defined by names (no evaluation in the preimage ring is possible).

Example:

```
  ring r1=32003,(x,y,z),dp;
  ideal i=x,y,z;
  ring r2=32003,(a,b),dp;
  map f=r1,a,b,a+b;
  // maps from r1 to r2,
  // x -> a
  // y -> b
  // z -> a+b
  f(i);
↦ _[1]=a
↦ _[2]=b
↦ _[3]=a+b
  // operations like f(i[1]) or f(i*i) are not allowed
  ideal i=f(i);
```

```
// objects in different rings may have the same name
map g   = r2,a2,b2;
map phi = g(f);
// composition of map f and g
// maps from r1 to r2,
// x -> a2
// y -> b2
// z -> a2+b2
phi(i);
↦ _[1]=a2
↦ _[2]=b2
↦ _[3]=a2+b2
```

See Section 5.1.37 [fetch], page 179; Section 4.5.2 [ideal expressions], page 79; Section 5.1.57 [imap], page 196; Section 4.11 [map], page 104; Section 4.19 [ring], page 125.

4.11.2 map expressions

A map expression is:

1. an identifier of type map
2. a function returning map
3. map expressions combined by composition using parentheses (`(`, `)`)

4.11.3 map operations

`( )` composition of maps. If, for example, `f` and `g` are maps, then `f(g)` is a map expression giving the composition $f \circ g$ of `f` and `g`,
provided the target ring of `g` is the basering of `f`.

map_expression `[` int_expressions `]`
is a map entry (the image of the corresponding variable)

Example:

```
ring r=0,(x,y),dp;
map f=r,y,x;    // the map f permutes the variables
f;
↦ f[1]=y
↦ f[2]=x
poly p=x+2y3;
f(p);
↦ 2x3+y
map g=f(f);    // the map g defined as  f^2 is the identity
g;
↦ g[1]=x
↦ g[2]=y
g(p) == p;
↦ 1
```

4.11.4 map related functions

`fetch` the identity map between rings (see Section 5.1.37 [fetch], page 179) `imap` a convenient map procedure for inclusions and projections of rings (see Section 5.1.57 [imap], page 196)

`preimage` preimage under a ring map (see Section 5.1.114 [preimage], page 236)

`subst` substitution of a ring variable (see Section 5.1.150 [subst], page 270)

See also the libraries Section D.4.2 [algebra_lib], page 741 and Section D.2.11 [ring_lib], page 735, which contain more functions, related to maps.

4.12 matrix

Objects of type matrix are matrices with polynomial entries. Like polynomials they can only be defined or accessed with respect to a basering. In order to compute with matrices having integer or rational entries, define a ring with characteristic 0 and at least one variable.

A matrix can be multiplied by and added to a poly; in this case the polynomial is converted into a matrix of the right size with the polynomial on the diagonal.

If A is a matrix then the assignment `module M=A;` or `module M=module(A);` creates a module generated by the columns of A. Note that the trailing zero columns of A may be deleted by module operations with M.

4.12.1 matrix declarations

Syntax: `matrix` name`[`rows`]` `[`cols`]` `=` list_of_poly_expressions `;`
`matrix` name = matrix_expression `;`

Purpose: defines a matrix (of polynomials).

The given poly_list fills up the matrix beginning with the first row from the left to the right, then the second row and so on. If the poly_list contains less than rows*cols elements, the matrix is filled up with zeros; if it contains more elements, then only the first rows*cols elements are used. If the right-hand side is a matrix expression the matrix on the left-hand side gets the same size as the right-hand side, otherwise the size is determined by the left-hand side. If the size is omitted a 1x1 matrix is created.

Default: 0 (1 x 1 matrix)

Example:

```
  int ro = 3;
  ring r = 32003,(x,y,z),dp;
  poly f=xyz;
  poly g=z*f;
  ideal i=f,g,g^2;
  matrix m[ro][3] = x3y4, 0, i, f ; // a 3 x 3 matrix
  m;
↦ m[1,1]=x3y4
↦ m[1,2]=0
↦ m[1,3]=xyz
↦ m[2,1]=xyz2
↦ m[2,2]=x2y2z4
↦ m[2,3]=xyz
↦ m[3,1]=0
↦ m[3,2]=0
↦ m[3,3]=0
  print(m);
```

```
↦ x3y4,0,     xyz,
↦ xyz2,x2y2z4,xyz,
↦ 0,   0,     0
  matrix A;   // the 1 x 1 zero matrix
  matrix B[2][2] = m[1..2, 2..3]; //defines a submatrix
  print(B);
↦ 0,     xyz,
↦ x2y2z4,xyz
  matrix C=m; // defines C as a 3 x 3 matrix equal to m
  print(C);
↦ x3y4,0,     xyz,
↦ xyz2,x2y2z4,xyz,
↦ 0,   0,     0
```

4.12.2 matrix expressions

A matrix expression is:

1. an identifier of type matrix
2. a function returning matrix
3. matrix expressions combined by the arithmetic operations `+`, `-` or `*`
4. a type cast to matrix (see Section 4.12.3 [matrix type cast], page 108)

Example:

```
  ring r=0,(x,y),dp;
  poly f= x3y2 + 2x2y2 +2;
  matrix H = jacob(jacob(f));    // the Hessian of f
  matrix mc = coef(f,y);
  print(mc);
↦ y2,    1,
↦ x3+2x2,2
  module MD = [x+y,1,x],[x+y,0,y];
  matrix M = MD;
  print(M);
↦ x+y,x+y,
↦ 1,  0,
↦ x,  y
```

4.12.3 matrix type cast

Syntax: `matrix` (expression)
`matrix` (expression, int_n, int_m)

Type: matrix

Purpose: Converts expression to a matrix, where expression must be of type int, intmat, intvec, number, poly, ideal, vector, module, or matrix. If int_n and int_m are supplied, then they specify the dimension of the matrix. Otherwise, the size (resp. dimensions) of the matrix is determined by the size (resp. dimensions) of the expression.

Example:

```
  ring r=32003,(x,y,z),dp;
  matrix(x);
```

```
↦ _[1,1]=x
  matrix(x, 1, 2);
↦ _[1,1]=x
↦ _[1,2]=0
  matrix(intmat(intvec(1,2,3,4), 2, 2));
↦ _[1,1]=1
↦ _[1,2]=2
↦ _[2,1]=3
↦ _[2,2]=4
  matrix(_, 2, 3);
↦ _[1,1]=1
↦ _[1,2]=2
↦ _[1,3]=0
↦ _[2,1]=3
↦ _[2,2]=4
↦ _[2,3]=0
  matrix(_, 2, 1);
↦ _[1,1]=1
↦ _[2,1]=3
```

See Section 3.5.5 [Type conversion and casting], page 45; Section 4.7.3 [intmat type cast], page 90; Section 4.12 [matrix], page 107.

4.12.4 matrix operations

`+` addition with matrix or poly; the polynomial is converted into a diagonal matrix

`-` negation or subtraction with matrix or poly (the first operand is expected to be a matrix); the polynomial is converted into a diagonal matrix

`*` multiplication with matrix or poly; the polynomial is converted into a diagonal matrix

`/` division by poly

`==`, `<>`, `!=` comparators

matrix_expression `[` int_expression`,` int_expression `]`
: is a matrix entry, where the first index indicates the row and the second the column

Example:

```
  ring r=32003,x,dp;
  matrix A[3][3] = 1,3,2,5,0,3,2,4,5; // define a matrix
  print(A); // nice printing of small matrices
↦ 1,3,2,
↦ 5,0,3,
↦ 2,4,5
  A[2,3];   // matrix entry
↦ 3
  A[2,3] = A[2,3] + 1; // change entry
  A[2,1..3] = 1,2,3;   // change 2nd row
  print(A);
↦ 1,3,2,
↦ 1,2,3,
↦ 2,4,5
  matrix E[3][3]; E = E + 1;  // the unit matrix
```

```
  matrix B =x*E - A;
  print(B);
↦ x-1,-3, -2,
↦ -1, x-2,-3,
↦ -2, -4, x-5
  // the same (but x-A does not work):
  B = -A+x;
  print(B);
↦ x-1,-3, -2,
↦ -1, x-2,-3,
↦ -2, -4, x-5
  det(B);        // the characteristic polynomial of A
↦ x3-8x2-2x-1
  A*A*A - 8 * A*A - 2*A == E;  // Cayley-Hamilton
↦ 1
  vector v =[x,-1,x2];
  A*v; // multiplication of matrix and vector
↦ _[1,1]=2x2+x-3
↦ _[2,1]=3x2+x-2
↦ _[3,1]=5x2+2x-4
  matrix m[2][2]=1,2,3;
  print(m-transpose(m));
↦ 0,-1,
↦ 1,0
```

4.12.5 matrix related functions

`bareiss` Gauss-Bareiss algorithm (see Section 5.1.3 [bareiss], page 156)

`coef` matrix of coefficients and monomials (see Section 5.1.11 [coef], page 162)

`coeffs` matrix of coefficients (see Section 5.1.12 [coeffs], page 162)

`det` determinant (see Section 5.1.22 [det], page 169)

`diff` partial derivative (see Section 5.1.23 [diff], page 169)

`jacob` Jacobi matrix (see Section 5.1.64 [jacob], page 200)

`koszul` Koszul matrix (see Section 5.1.71 [koszul], page 205)

`lift` lift-matrix (see Section 5.1.78 [lift], page 209)

`liftstd` standard basis and transformation matrix computation (see Section 5.1.79 [liftstd], page 209)

`minor` set of minors of a matrix (see Section 5.1.90 [minor], page 218)

`ncols` number of columns (see Section 5.1.101 [ncols], page 227)

`nrows` number of rows (see Section 5.1.104 [nrows], page 229)

`print` nice print format (see Section 5.1.117 [print], page 238)

`size` number of matrix entries (see Section 5.1.140 [size], page 260)

`subst` substitute a ring variable (see Section 5.1.150 [subst], page 270)

`trace` trace of a matrix (see Section 5.1.153 [trace], page 276)

`transpose`
transposed matrix (see Section 5.1.154 [transpose], page 276)

`wedge` wedge product (see Section 5.1.166 [wedge], page 282)

See also the library Section D.3.1 [matrix_lib], page 738, which contains more matrix-related functions.

4.13 module

Modules are submodules of a free module over the basering with basis `gen(1)`, `gen(2)`, ... They are represented by lists of vectors which generate the submodule. Like vectors they can only be defined or accessed with respect to a basering.

If R is the basering, and M is a submodule of R^n

generated by vectors $v_1, \ldots, v_k$, then $v_1, \ldots, v_k$

may be considered as the generators of relations of R^n/M between the canonical generators `gen(1)`,. . .,`gen(n)`. Hence any finitely generated R -module can be represented in SINGULAR by its module of relations. The assignments `module M=v1,...,vk; matrix A=M;` create the presentation matrix of size n × k for R^n/M , i.e., the columns of A are the vectors $v_1, \ldots, v_k$ which generate M (cf. Section B.1 [Representation of mathematical objects], page 695).

4.13.1 module declarations

Syntax: `module` name = list_of_vector_expressions `;`
`module` name = module_expression `;`

Purpose: defines a module.

Default: [0]

Example:

```
  ring r=0,(x,y,z),(c,dp);
  vector s1 = [x2,y3,z];
  vector s2 = [xy,1,0];
  vector s3 = [0,x2-y2,z];
  poly   f  = xyz;
  module m = s1, s2-s1,f*(s3-s1);
  m;
↦ m[1]=[x2,y3,z]
↦ m[2]=[-x2+xy,-y3+1,-z]
↦ m[3]=[-x3yz,-xy4z+x3yz-xy3z]
  // show m in matrix format (columns generate m)
  print(m);
↦ x2,-x2+xy,-x3yz,
↦ y3,-y3+1, -xy4z+x3yz-xy3z,
↦ z, -z,    0
```

4.13.2 module expressions

A module expression is:

1. an identifier of type module
2. a function returning module
3. module expressions combined by the arithmetic operation `+`

4. multiplication of a module expression with an ideal or a poly expression: `*`
5. a type cast to module

See Section 3.5.5 [Type conversion and casting], page 45; Section 4.5 [ideal], page 79; Section 4.16 [poly], page 118; Section 4.21 [vector], page 131.

4.13.3 module operations

`+` addition (concatenation of the generators and simplification)

`*` multiplication with ideal or poly (but not ‘module‘ * ‘module‘!)

module_expression `[` int_expression `,` int_expression `]`
: is a module entry, where the first index indicates the row and the second the column

module_expressions `[` int_expression `]`
: is a vector, where the index indicates the column (generator)

Example:

```
  ring r=0,(x,y,z),dp;
  module m=[x,y],[0,0,z];
  print(m*(x+y));
↦ x2+xy,0,
↦ xy+y2,0,
↦ 0,    xz+yz
  // this is not distributive:
  print(m*x+m*y);
↦ x2,0, xy,0,
↦ xy,0, y2,0,
↦ 0, xz,0, yz
```

4.13.4 module related functions

`coeffs` matrix of coefficients (see Section 5.1.12 [coeffs], page 162)

`degree` multiplicity, dimension and codimension of the module of leading terms (see Section 5.1.19 [degree], page 167)

`diff` partial derivative (see Section 5.1.23 [diff], page 169)

`dim` Krull dimension of free module over the basering modulo the module of leading terms (see Section 5.1.24 [dim], page 170)

`eliminate`
: elimination of variables (see Section 5.1.27 [eliminate], page 173)

`freemodule`
: the free module of given rank (see Section 5.1.45 [freemodule], page 185)

`fres` free resolution of a standard basis (see Section 5.1.46 [fres], page 185)

`groebner` Groebner basis computation (a wrapper around `std,stdhilb,stdfglm`,...) (see [groebner], page 721)

`hilb` Hilbert function of a standard basis (see Section 5.1.54 [hilb], page 193)

`homog` homogenization with respect to a variable (see Section 5.1.55 [homog], page 194) `interred` interreduction of a module (see Section 5.1.62 [interred], page 199)

`intersect`
module intersection (see Section 5.1.63 [intersect], page 200)

`jet` Taylor series up to a given order (see Section 5.1.66 [jet], page 202)

`kbase` vector space basis of free module over the basering modulo the module of leading terms (see Section 5.1.67 [kbase], page 203)

`lead` initial module (see Section 5.1.73 [lead], page 206)

`lift` lift-matrix (see Section 5.1.78 [lift], page 209)

`liftstd` standard basis and transformation matrix computation (see Section 5.1.79 [liftstd], page 209)

`lres` free resolution (see Section 5.1.81 [lres], page 212)

`minbase` minimal generating set of a homogeneous ideal, resp. module, or an ideal, resp. module, over a local ring

`modulo` represents $(h1 + h2)/h1 = h2/(h1 \cap h2)$ (see Section 5.1.92 [modulo], page 221)

`mres` minimal free resolution of an ideal resp. module w.r.t. a minimal set of generators of the given module (see Section 5.1.96 [mres], page 223)

`mult` multiplicity, resp. degree, of the module of leading terms (see Section 5.1.98 [mult], page 224)

`nres` computation of a free resolution of an ideal resp. module M which is minimized from the second free module on (see Section 5.1.103 [nres], page 228)

`ncols` number of columns (see Section 5.1.101 [ncols], page 227)

`nrows` number of rows (see Section 5.1.104 [nrows], page 229)

`print` nice print format (see Section 5.1.117 [print], page 238)

`prune` minimization of the embedding into a free module (see Section 5.1.119 [prune], page 241)

`qhweight` quasihomogeneous weights of an ideal, resp. module (see Section 5.1.120 [qhweight], page 242)

`quotient` module quotient (see Section 5.1.123 [quotient], page 243)

`reduce` normalform with respect to a standard basis (see Section 5.1.127 [reduce], page 246)

`res` free resolution of an ideal, resp. module, but not changing the given ideal, resp. module (see [res], page 721)

`simplify` simplification of a set of vectors (see Section 5.1.139 [simplify], page 258)

`size` number of non-zero generators (see Section 5.1.140 [size], page 260)

`sortvec` permutation for sorting ideals/modules (see Section 5.1.142 [sortvec], page 261)

`sres` free resolution of a standard basis (see Section 5.1.145 [sres], page 264)

`std` standard basis computation (see Section 5.1.147 [std], page 266, Section 5.1.79 [liftstd], page 209)

`subst` substitution of a ring variable (see Section 5.1.150 [subst], page 270)

`syz` computation of the first syzygy module (see Section 5.1.152 [syz], page 275) `vdim` vector space dimension of free module over the basering modulo module of leading terms (see Section 5.1.163 [vdim], page 280)

`weight` "optimal" weights (see Section 5.1.167 [weight], page 283)

4.14 number

Numbers are elements from the coefficient ring (or ground ring). They can only be defined or accessed with respect to a basering which determines the coefficient field. See Section 4.19.2 [ring declarations], page 125 for declarations of coefficient fields.

Warning: Beware of the special meaning of the letter `e` (immediately following a sequence of digits) if the field is real (or complex), Section 6.4 [Miscellaneous oddities], page 308.

4.14.1 number declarations

Syntax: `number` name = number_expression `;`

Purpose: defines a number.

Default: 0

Note: Numbers may only be declared w.r.t. the coefficient field of the current basering, i.e., a ring has to be defined prior to any number declaration. See Section 3.3 [Rings and orderings], page 30 for a list of the available coefficient fields.

Example:

```
  // finite field Z/p, p<= 32003
  ring r = 32003,(x,y,z),dp;
  number n = 4/6;
  n;
↦ -10667
  // finite field GF(p^n), p^n <= 32767
  // z is a primitive root of the minimal polynomial
  ring rg= (7^2,z),x,dp;
  number n = 4/9+z;
  n;
↦ z38
  // the rational numbers
  ring r0 = 0,x,dp;
  number n = 4/6;
  n;
↦ 2/3
  // algebraic extensions of Z/p or Q
  ring ra=(0,a),x,dp;
  minpoly=a^2+1;
  number n=a3+a2+2a-1;
  n;
↦ (a-2)
  a^2;
↦ -1
  // transcedental extensions of Z/p or Q
  ring rt=(0,a),x,dp;
  number n=a3+a2+2a-1;
  n;
```

```
↦ (a3+a2+2a-1)
  a^2;
↦ (a2)
  // machine floating point numbers, single precision
  ring R_0=real,x,dp;
  number n=4/6;
  n;
↦ (6.667e-01)
  n=0.25e+2;
  n;
↦ (2.500e+01)
  // floating point numbers, arbitrary prescribed precision
  ring R_1=(real,50),x,dp;
  number n=4.0/6;
  n;
↦ 0.66666666666666666666666666666666666666666666666667
  n=0.25e+2;
  n;
↦ 25
  // floating point complex numbers, arbitrary prescribed precision
  // the third parameter gives the name of the imaginary unit
  ring R_2=(complex,50,i),x,dp;
  number n=4.0/6;
  n;
↦ 0.66666666666666666666666666666666666666666666666667
  n=0.25e+2*i+n;
  n;
↦ (0.66666666666666666666666666666666666666666666666667+i*25)
```

4.14.2 number expressions

A number expression is:

1. a rational number (there are NO spaces allowed inside a rational number, see Section 4.6.2 [int expressions], page 84)
2. a floating point number (if the coefficient field is `real`): `<digits>.<digits>e<sign><digits>`
3. an identifier of type number
4. a function returning number
5. an int expression (see Section 3.5.5 [Type conversion and casting], page 45)
6. number expressions combined by the arithmetic operations `+`, `-`, `*`, `/`, `^`, or `**`.
7. a type cast to number

Example:

```
  // the following expressions are in any ring int expressions
  2 / 3;
↦ // ** int division with ‘/‘: use ‘div‘ instead in line >>  2 / 3;<<
↦ 0
  4/ 8;
↦ // ** int division with ‘/‘: use ‘div‘ instead in line >>  4/ 8;<<
↦ 0
```

```
  2 /2;   // the notation of / for div might change in the future
↦ // ** int division with ‘/‘: use ‘div‘ instead in line >>  2 /2;   // the\
    notation of / for div might change in the future<<
↦ 1
  ring r0=0,x,dp;
  2/3, 4/8, 2/2 ; // are numbers
↦ 2/3 1/2 1

  poly f = 2x2 +1;
  leadcoef(f);
↦ 2
  typeof(_);
↦ number
  ring rr =real,x,dp;
  1.7e-2; 1.7e+2; // are valid (but  1.7e2 not), if the field is ‘real‘
↦ (1.700e-02)
↦ (1.700e+02)
  ring rp = (31,t),x,dp;
  2/3, 4/8, 2/2 ; // are numbers
↦ 11 -15 1
  poly g = (3t2 +1)*x2 +1;
  leadcoef(g);
↦ (3t2+1)
  typeof(_);
↦ number
  par(1);
↦ (t)
  typeof(_);
↦ number
```

See Section 3.5.5 [Type conversion and casting], page 45; Section 4.19 [ring], page 125.

4.14.3 number operations

`+` addition

`-` negation or subtraction

`*` multiplication

`/` division

`%`, `mod` modulo

`^`, `**` power, exponentiation (by an integer)

`<=`, `>=`, `==`, `<>`
comparison

`mod` integer modulo (the remainder of the division `div`), always non-negative

Note: Quotient and exponentiation is only recognized as a number expression if it is already a number, see Section 6.4 [Miscellaneous oddities], page 308.
For the behavior of comparison operators in rings with ground field different from real or the rational numbers, see Section 4.6.5 [boolean expressions], page 87.

Example:

```
  ring r=0,x,dp;
  number n = 1/2 +1/3;
  n;
↦ 5/6
  n/2;
↦ 5/12
  1/2/3;
↦ 1/6
  1/2 * 1/3;
↦ 1/6
  n = 2;
  n^-2;
↦ 1/4
  // the following oddities appear here
  2/(2+3);
↦ // ** int division with `/`: use `div` instead in line >>  2/(2+3);<<
↦ 0
  number(2)/(2+3);
↦ 2/5
  2^-2; // for int's exponent must be non-negative
↦    ? exponent must be non-negative
↦    ? error occurred in or before ./examples/number_operations.sing line 1\
   2: `  2^-2; // for int's exponent must be non-negative`
  number(2)^-2;
↦ 1/4
  3/4>=2/5;
↦ 1
  2/6==1/3;
↦ 1
```

4.14.4 number related functions

`cleardenom`
: cancellation of denominators of numbers in polyomial and divide it by its content (see Section 5.1.9 [cleardenom], page 161)

`impart`
: imaginary part of a complex number, 0 otherwise (see Section 5.1.58 [impart], page 196, Section 5.1.129 [repart], page 248)

`numerator, denominator`
: the numerator/denominator of a rational number (see Section 5.1.105 [numerator], page 229, Section 5.1.21 [denominator], page 168)

`leadcoef`
: coefficient of the leading term (see Section 5.1.74 [leadcoef], page 207)

`par`
: n-th parameter of the basering (see Section 5.1.111 [par], page 235)

`pardeg`
: degree of a number in ring parameters (see Section 5.1.112 [pardeg], page 236)

`parstr`
: string form of ring parameters (see Section 5.1.113 [parstr], page 236)

`repart`
: real part of a complex number (see Section 5.1.58 [impart], page 196, Section 5.1.129 [repart], page 248)

4.15 package

The data type package is used to group identifiers into collections. It is mainly used as an internal means to avoid collisions of names of identifiers in libraries with variable names defined by the user. The most important package is the toplevel package, called `Top`. It contains all user defined identifiers as well as all user accessible library procedures. Identifiers which are local to a library are contained in a package whose name is obtained from the name of the library, where the first letter is converted to uppercase, the remaining ones to lowercase. Another reserved package name is `Current` which denotes the current package name in use. See also Section 3.8 [Libraries], page 55.

4.15.1 package declarations

Syntax: `package` name ;

Purpose: defines a package (Only relevant in very special situations).

Example:

```
  package Test;
  int Test::i;
  listvar();
  listvar(Test);
↦ // Test                                [0]  package Test (N)
↦ // ::i                                 [0]  int 0
  package dummy = Test;
  kill Test;
  listvar(dummy);
↦ // dummy                               [0]  package dummy (N)
↦ // ::i                                 [0]  int 0
```

4.15.2 package related functions

`exportto` transfer an identifier to the specified package (see Section 5.2.7 [exportto], page 287)

`importfrom`
generate a copy of an identifier from the specified package in the current package (see Section 5.2.10 [importfrom], page 291)

`listvar` list variables currently defined in a given package (see Section 5.1.80 [listvar], page 210)

`load` load a library or dynamic module (see Section 5.2.12 [load], page 293)

`LIB` load a library or dynamic module (see Section 5.1.77 [LIB], page 208)

4.16 poly

Polynomials are the basic data for all main algorithms in Singular. They consist of finitely many terms (coefficient*monomial) which are combined by the usual polynomial operations (see Section 4.16.2 [poly expressions], page 119). Polynomials can only be defined or accessed with respect to a basering which determines the coefficient type, the names of the indeterminates and the monomial ordering.

```
ring r=32003,(x,y,z),dp;
poly f=x3+y5+z2;
```

4.16.1 poly declarations

Syntax: `poly` name = poly_expression ;

Purpose: defines a polynomial.

Default: 0

Example:

```
ring r = 32003,(x,y,z),dp;
poly s1  = x3y2+151x5y+186xy6+169y9;
poly s2  = 1*x^2*y^2*z^2+3z8;
poly s3  = 5/4x4y2+4/5*x*y^5+2x2y2z3+y7+11x10;
int a,b,c,t=37,5,4,1;
poly f=3*x^a+x*y^(b+c)+t*x^a*y^b*z^c;
f;
↦ x37y5z4+3x37+xy9
short = 0;
f;
↦ x^37*y^5*z^4+3*x^37+x*y^9
```

Section 5.3.7 [short], page 299

4.16.2 poly expressions

A polynomial expression is (optional parts in square brackets):

1. a monomial (there are NO spaces allowed inside a monomial)

   `[coefficient] ring_variable [ exponent] [ring_variable [exponent] ...].`

   Monomials which contain an indexed ring variable must be built from `ring_variable` and `coefficient` with the operations `*` and `^`
2. an identifier of type poly
3. a function returning poly
4. polynomial expressions combined by the arithmetic operations `+`, `-`, `*`, `/`, or `^`
5. an int expression (see Section 3.5.5 [Type conversion and casting], page 45)
6. a type cast to poly

Example:

```
ring S=0,(x,y,z,a(1)),dp;
2x, x3, 2x2y3, xyz, 2xy2; //  are monomials
2*x, x^3, 2*x^2*y^3, x*y*z, 2*x*y^2; // are poly expressions
2*a(1); // is a valid polynomial expression (a(1) is a name of a variable),
        // but not 2a(1) (is a syntax error)
2*x^3;  // is a valid polynomial expression equal to 2x3 (a valid monomial)
        // but not equal to 2x^3 which will be interpreted as (2x)^3
        // since 2x is a monomial
  ring r=0,(x,y),dp;
  poly f = 10x2y3 +2x2y2-2xy+y -x+2;
  lead(f);
↦ 10x2y3
  leadmonom(f);
↦ x2y3
  simplify(f,1);     // normalize leading coefficient
```

```
↦ x2y3+1/5x2y2-1/5xy-1/10x+1/10y+1/5
  poly g = 1/2x2 + 1/3y;
  cleardenom(g);
↦ 3x2+2y
  int i = 102;
  poly(i);
↦ 102
  typeof(_);
↦ poly
```

See Section 3.5.5 [Type conversion and casting], page 45; Section 4.19 [ring], page 125.

4.16.3 poly operations

`+` addition

`-` negation or subtraction

`*` multiplication

`/`, `div` division by a polynomial, ignoring the remainder (only implemented for polynomials over QQ, ZZ/p and field extensions of them)
(See also Section 5.1.123 [quotient], page 243,Section 5.1.25 [division], page 171,Section 5.1.127 [reduce], page 246)

`%`, `mod` the remainder from the division by a polynomial (only implemented for polynomials over QQ, ZZ/p and field extensions of them)
(See also Section 5.1.123 [quotient], page 243,Section 5.1.25 [division], page 171,Section 5.1.127 [reduce], page 246)

`^`, `**` power by a positive integer

`<`, `<=`, `>`, `>=`, `==`, `<>`
comparators (considering leading monomials w.r.t. monomial ordering)

poly_expression `[` intvec_expression `]`
the sum of monomials at the indicated places w.r.t. the monomial ordering

Example:

```
  ring R=0,(x,y),dp;
  poly f = x3y2 + 2x2y2 + xy - x + y + 1;
  f;
↦ x3y2+2x2y2+xy-x+y+1
  f + x5 + 2;
↦ x5+x3y2+2x2y2+xy-x+y+3
  f * x2;
↦ x5y2+2x4y2+x3y-x3+x2y+x2
  (x+y)/x;
↦ 1
  f/3x2;
↦ 1/3xy2+2/3y2
  x5 > f;
↦ 1
  x<=y;
↦ 0
  x>y;
```

```
↦ 1
  ring r=0,(x,y),ds;
  poly f = fetch(R,f);
  f;
↦ 1-x+y+xy+2x2y2+x3y2
  x5 > f;
↦ 0
  f[2..4];
↦ -x+y+xy
  size(f);
↦ 6
  f[size(f)+1]; f[-1];    // monomials out of range are 0
↦ 0
↦ 0
  intvec v = 6,1,3;
  f[v];          // the polynom built from the 1st, 3rd and 6th monomial of f
↦ 1+y+x3y2
```

4.16.4 poly related functions

`cleardenom`
: cancellation of denominators of numbers in polynomial and divide it by its content (see Section 5.1.9 [cleardenom], page 161; [content], page 733)

`coef` matrix of coefficients and monomials (see Section 5.1.11 [coef], page 162)

`coeffs` matrix of coefficients (see Section 5.1.12 [coeffs], page 162)

`deg` degree (see Section 5.1.18 [deg], page 167)

`diff` partial derivative (see Section 5.1.23 [diff], page 169)

`extgcd` Bezout representation of gcd (see Section 5.1.32 [extgcd], page 176)

`factorize`
: factorization of polynomial (see Section 5.1.35 [factorize], page 178)

`finduni` univariate polynomials in a zero-dimensional ideal (see Section 5.1.42 [finduni], page 182)

`gcd` greatest common divisor (see Section 5.1.48 [gcd], page 187)

`homog` homogenization (see Section 5.1.55 [homog], page 194)

`jacob` ideal, resp. matrix, of all partial derivatives (see Section 5.1.64 [jacob], page 200)

`lead` leading term (see Section 5.1.73 [lead], page 206)

`leadcoef` coefficient of the leading term (see Section 5.1.74 [leadcoef], page 207)

`leadexp` the exponent vector of the leading monomial (see Section 5.1.75 [leadexp], page 207)

`leadmonom`
: leading monomial (see Section 5.1.76 [leadmonom], page 208)

`jet` monomials of degree at most k (see Section 5.1.66 [jet], page 202)

`ord` degree of the leading monomial (see Section 5.1.109 [ord], page 235)

`qhweight` quasihomogeneous weights (see Section 5.1.120 [qhweight], page 242) `reduce` normal form with respect to a standard base (see Section 5.1.127 [reduce], page 246)

`rvar` test for ring variable (see Section 5.1.135 [rvar], page 254)

`simplify` normalization of a polynomial (see Section 5.1.139 [simplify], page 258)

`size` number of monomials (see Section 5.1.140 [size], page 260)

`subst` substitution of a ring variable (see Section 5.1.150 [subst], page 270)

`trace` trace of a matrix (see Section 5.1.153 [trace], page 276)

`var` the indicated variable of the ring (see Section 5.1.160 [var], page 279)

`varstr` variable(s) in string form (see Section 5.1.162 [varstr], page 280)

4.17 proc

Procedures are sequences of Singular commands in a special format. They are used to extend the set of Singular commands with user defined commands. Once a procedure is defined it can be used as any other Singular command. Procedures may be defined by either typing them on the command line or by loading them from a file. For a detailed description on the concept of procedures in Singular see Section 3.7 [Procedures], page 51. A file containing procedure definitions which comply with certain syntax rules is called a library. Such a file is loaded using the command `LIB`. For more information on libraries see Section 3.8 [Libraries], page 55.

4.17.1 proc declaration

Syntax:
[`static`] `proc` proc_name [(<parameter_list>)
[<help_string>]
`{`
  <procedure_body>
`}`
[`example`
`{`
  <sequence_of_commands>
`}`]

Purpose: Defines a new function, the `proc` proc_name. Once loaded in a Singular session, the information provided in the help string will be displayed upon entering `help proc_name;`, while the `example` section will be executed upon entering `example proc_name;`. See Section 3.7.2 [Parameter list], page 53, Section 3.7.3 [Help string], page 54, and the example in Section 3.8.6 [Procedures in a library], page 58.
The help string, the parameter list, and the example section are optional. They are, however, mandatory for the procedures listed in the header of a library. The help string is ignored and no example section is allowed if the procedure is defined interactively, i.e., if it is not loaded from a file by the `LIB` or `load` command (see Section 5.1.77 [LIB], page 208 and see Section 5.2.12 [load], page 293).
In the body of a library, each procedure not meant to be accessible by users should be declared static. See Section 3.8.6 [Procedures in a library], page 58.

Example:

```
proc milnor_number (poly p)
{
  ideal i= std(jacob(p));
```

```
    int m_nr=vdim(i);
    if (m_nr<0)
    {
      "// not an isolated singularity";
    }
    return(m_nr);         // the value of m_nr is returned
  }
  ring r1=0,(x,y,z),ds;
  poly p=x^2+y^2+z^5;
  milnor_number(p);
↦ 4
```

See Section 5.1.77 [LIB], page 208; Section 3.8 [Libraries], page 55; Section 5.2.1 [apply], page 285.

4.17.2 proc expression

Syntax: variable_name `->` `{` expression(s) `}`

Purpose: Defines a new function, within `apply` or for assigning.

Example:

```
  apply(1..3,x->{x**2});
↦ 1 4 9
```

See Section 5.2.1 [apply], page 285; Section 4.17 [proc], page 122.

4.17.3 procs with different argument types

Syntax: `branchTo` `(` string_expression `,` ... proc_name `)`

Purpose: branch to the given procedure if the argument types matches the types given as strings (which may be empty - matching the empty argument list). The main procedure (`p` in the example) must be defined without an argemnt list, and `branchTo` statement must be the first statement within the procedure body.

Example:

```
  proc p1(int i) { "int:",i; }
  proc p21(string s) { "string:",s; }
  proc p22(string s1, string s2) { "two strings:",s1,s2; }
  proc p()
  { branchTo("int",p1);
    branchTo("string","string",p22);
    branchTo("string",p21);
    ERROR("not defined for these argument types");
  }
  p(1);
↦ int: 1
  p("hu");
↦ string: hu
  p("ha","ha");
↦ two strings: ha ha
  p(1,"hu");
↦    ? not defined for these argument types
↦    ? leaving ::p
```

See Section 4.17 [proc], page 122.

4.18 resolution

The type resolution is intended as an intermediate representation which internally retains additional information obtained during computation of resolutions. It furthermore enables the use of partial results to compute, for example, Betti numbers or minimal resolutions. Like ideals and modules, a resolution can only be defined w.r.t. a basering (see Section C.3 [Syzygies and resolutions], page 703).

Note: To access the elements of a resolution, it has to be assigned to a list. This assignment also completes computations and may therefore take time, (resp. an access directly with the brackets `[ , ]` causes implicitly a cast to a list).

4.18.1 resolution declarations

Syntax: `resolution` name = resolution_expression ;

Purpose: defines a resolution.

Default: none

Example:

```
  ring R;
  ideal i=z2,x;
  resolution re=res(i,0);
  re;
↦  1      2      1
↦ R <--  R <--  R
↦
↦ 0      1      2
↦
  betti(re);
↦ 1,1,0,
↦ 0,1,1
  list l = re;
  l;
↦ [1]:
↦    _[1]=x
↦    _[2]=z2
↦ [2]:
↦    _[1]=-z2*gen(1)+x*gen(2)
↦ [3]:
↦    _[1]=0
```

4.18.2 resolution expressions

A resolution expression is:

1. an identifier of type resolution
2. a function returning a resolution
3. a type cast to resolution from a list of ideals, resp. modules..

See Section 3.5.5 [Type conversion and casting], page 45.

4.18.3 resolution related functions

`betti` Betti numbers of a resolution (see Section 5.1.4 [betti], page 157)

`fres` free resolution of a standard basis (see Section 5.1.46 [fres], page 185)

`lres` free resolution (see Section 5.1.81 [lres], page 212)

`minres` minimize a free resolution (see Section 5.1.91 [minres], page 220)

`mres` minimal free resolution of an ideal, resp. module and a minimal set of generators of the given ideal, resp. module (see Section 5.1.96 [mres], page 223)

`res` free resolution of an ideal, resp. module, but not changing the given ideal, resp. module (see [res], page 721)

`sres` free resolution of a standard basis (see Section 5.1.145 [sres], page 264)

4.19 ring

Rings are used to describe properties of polynomials, ideals etc. Almost all computations in Singular require a basering. For a detailed description of the concept of rings see Section 3.3 [Rings and orderings], page 30.

4.19.1 qring

Singular offers the opportunity to calculate in quotient rings (factor rings), i.e., rings modulo an ideal. The ideal has to be given as a standard basis. For a detailed description of the concept of rings and quotient rings see Section 3.3 [Rings and orderings], page 30. Beside the construction, an object describing a quotient ring is of type `ring`.

See Section 4.19.5 [qring declaration], page 127.

4.19.2 ring declarations

Syntax: `ring` name `=` `(` coefficients `)`, `(` names_of_ring_variables `)`, `(` ordering `);` or
`ring` name `=` cring `[` names_of_ring_variables `]`

Default: `(ZZ/32003)[x,y,z]`

Purpose: declares a ring and sets it as the actual basering. The second form sets the ordering to `(dp,C)`.

For the second form: `cring` stands currently for `QQ` (the rationals), `ZZ` (the integers) or `(ZZ/m)` (the field (m prime and <2147483648) resp. ring of the integers modulo m).

The coefficients for the first form are given by one of the following:

1. a `cring` as given above
2. a non-negative int_expression less or equal 2147483647.
3. an expression_list of an int_expression and one or more names.
4. the name `real`
5. an expression_list of the name `real` and an int_expression.
6. an expression_list of the name `complex`, an optional int_expression and a name.
7. an expression_list of the name `integer`.
8. an expression_list of the name `integer` and following int_expressions.

9. an expression_list of the name `integer` and two int_expressions.

For the definition of the 'coefficients', see Section 3.3 [Rings and orderings], page 30.

'names_of_ring_variables' must be a list of names or (multi-)indexed names.

'ordering' is a list of block orderings where each block ordering is either

1. `lp`, `dp`, `Dp`, `rp`, `ls`, `ds`, `Ds`, or `rs` optionally followed by a size parameter in parentheses.
2. `wp`, `Wp`, `ws`, `Ws`, or `a` followed by a weight vector given as an intvec_expression in parentheses.
3. `M` followed by an intmat_expression in parentheses.
4. `c` or `C`.

For the definition of the orderings, see Section 3.3.3 [Term orderings], page 34, Section B.2 [Monomial orderings], page 696.

If one of coefficients, names_of_ring_variables, and ordering consists of only one entry, the parentheses around this entry may be omitted.

See also Section 3.3.1 [Examples of ring declarations], page 31; Section 4.19 [ring], page 125; Section 5.1.133 [ringlist], page 251.

4.19.3 ring related functions

`charstr` description of the coefficient field of a ring (see Section 5.1.7 [charstr], page 160)

`keepring` move ring to next upper level (see Section 5.2.11 [keepring], page 293)

`npars` number of ring parameters (see Section 5.1.102 [npars], page 228)

`nvars` number of ring variables (see Section 5.1.106 [nvars], page 230)

`ordstr` monomial ordering of a ring (see Section 5.1.110 [ordstr], page 235)

`parstr` names of all ring parameters or the name of the n-th ring parameter (see Section 5.1.113 [parstr], page 236)

`qring` quotient ring (see Section 4.19.1 [qring], page 125)

`ringlist` decomposition of a ring into a list of its components (see Section 5.1.133 [ringlist], page 251)

`setring` setting of a new basering (see Section 5.1.137 [setring], page 255)

`varstr` names of all ring variables or the name of the n-th ring variable (see Section 5.1.162 [varstr], page 280)

4.19.4 ring operations

`+` construct a new ring $k[X,Y]$ from $k_1[X]$ and $k_2[Y]$. (The sets of variables must be distinct).

`==,<>` compare two rings

Note: Concerning the ground fields k_1 and k_2 take the following guide lines into consideration:

- Neither k_1 nor k_2 may be R or C .
- If the characteristic of k_1 and k_2 differs, then one of them must be Q .
- At most one of k_1 and k_2 may have parameters.
- If one of k_1 and k_2 is an algebraic extension of Z/p it may not be defined by a `charstr` of type `(p^n,a)`.

Example:

```
ring R1=0,(x,y),dp;
ring R2=32003,(a,b),dp;
def R=R1+R2;
R;
↦ // coefficients: ZZ/32003
↦ // number of vars : 4
↦ //        block   1 : ordering dp
↦ //                  : names    x y
↦ //        block   2 : ordering dp
↦ //                  : names    a b
↦ //        block   3 : ordering C
```

Section D.2.11 [ring_lib], page 735

4.19.5 qring declaration

Syntax: `qring` name = ideal_expression ;

Default: none

Purpose: declares a quotient ring as the basering modulo ideal_expression and sets it as current basering.

Operations based on standard bases (e.g. `std`,`groebner`, etc., `reduce`) and functions which require a standard basis (e.g. `dim`,`hilb`, etc.) operated with the residue classes; all others on the polynomial objects.

Example:

```
ring r=0,(x,y,z),dp;
ideal i=xy;
qring q=std(i);
basering;
↦ // coefficients: QQ
↦ // number of vars : 3
↦ //        block   1 : ordering dp
↦ //                  : names    x y z
↦ //        block   2 : ordering C
↦ // quotient ring from ideal
↦ _[1]=xy
// simplification is not immediate:
(x+y)^2;
↦ x2+2xy+y2
reduce(_,std(0));
↦ x2+y2
// polynomial and residue class:
ring R=0,(x,y),dp;
qring Q=std(y);
poly p1=x;
poly p2=x+y;
// comparing polynomial objects:
p1==p2;
↦ 0
// comparing residue classes:
reduce(p1,std(0))==reduce(p2,std(0));
↦ 1
```

4.20 string

Variables of type `string` are used for output (almost every type can be "converted" to `string`) and for creating new commands at runtime see Section 5.1.31 [execute], page 175. They are also return values of certain interpreter related functions (see Section 5.1 [Functions], page 154). String constants consist of a sequence of ANY characters (including newline!) between a starting `"` and a closing `"`. There is also a string constant `newline`, which is the newline character. The `+` sign `"adds"` strings, `""` is the empty string (hence strings form a semigroup). Strings may be used to comment the output of a computation or to give it a nice format. Strings may also be used for intermediate conversion of one type into another.

```
  string s="Hi";
  string s1="a string with new line at the end"+newline;
  string s2="another string with new line at the end
  ";
  s;s1;s2;
↦ Hi
↦ a string with new line at the end
↦
↦ another string with new line at the end
↦
  ring r; ideal i=std(ideal(x,y^3));
  "dimension of i =",dim(i),", multiplicity of i =",mult(i);
↦ dimension of i = 1 , multiplicity of i = 3
  "dimension of i = "+string(dim(i))+", multiplicity of i = "+string(mult(i));
↦ dimension of i = 1, multiplicity of i = 3
  "a"+"b","c";
↦ ab c
```

A comma between two strings makes an expression list out of them (such a list is printed with a separating blank in between), while a `+` concatenates strings.

4.20.1 string declarations

Syntax: `string` name = string_expression ;
`string` name = list_of_string_expressions ;

Purpose: defines a string variable.

Default: `""` (the empty string)

Example:

```
  string s1="Now I know";
  string s2="how to encode a \" in a string...";
  string s=s1+" "+s2; // concatenation of 3 strings
  s;
↦ Now I know how to encode a " in a string...
  s1,s2;   // 2 strings, separated by a blank in the output:
↦ Now I know how to encode a " in a string...
```

4.20.2 string expressions

A string expression is:

1. a sequence of characters between two unescaped quotes (`"`)
2. an identifier of type string

3. a function returning string
4. a substring (using the bracket operator)
5. a type cast to string (see Section 4.20.3 [string type cast], page 129)
6. string expressions combined by the operation +.

Example:

```
// string_expression[start, length] : a substring
// (possibly filled up with blanks)
// the substring of s starting at position 2
// with a length of 4
string s="123456";
s[2,4];
↦ 2345
"abcd"[2,2];
↦ bc
// string_expression[position] : a character from a string
s[3];
↦ 3
// string_expression[position..position] :
// a substring starting at the first position up to the second
// given position
s[2..4];
↦ 2 3 4
// a function returning a string
typeof(s);
↦ string
```

See Section 3.5.5 [Type conversion and casting], page 45

4.20.3 string type cast

Syntax: `string` (expression [, expression_2, ... expression_n])

Type: string

Purpose: Converts each expression to a string, where expression can be of any type. The concatenated string of all converted expressions is returned.

The elements of intvec, intmat, ideal, module, matrix, and list, are separated by a comma. No newlines are inserted.
Not defined elements of a list are omitted.
For link, the name of the link is used.
For map, the ideal defining the mapping is converted.

Note: When applied to a list, elements of type intvec, intmat, ideal, module, matrix, and list become indistinguishable.

Example:

```
  string("1+1=", 2);
↦ 1+1=2
  string(intvec(1,2,3,4));
↦ 1,2,3,4
  string(intmat(intvec(1,2,3,4), 2, 2));
```

```
↦ 1,2,3,4
  ring r;
  string(r);
↦ (ZZ/32003),(x,y,z),(dp(3),C)
  string(ideal(x,y));
↦ x,y
  qring R = std(ideal(x,y));
  string(R);
↦ (ZZ/32003),(x,y,z),(dp(3),C)
  map phi = r, ideal(x,z);
  string(phi);
↦ x,z
  list l;
  string(l);
↦
  l[3] = 1;
  string(l); // notice that l[1],l[2] are omitted
↦ 1
  l[2] = l;
  l;
↦ [2]:
↦    [3]:
↦       1
↦ [3]:
↦    1
  string(l); // notice that lists of list is flattened
↦ 1,1
  l[1] = intvec(1,2,3);
  l;
↦ [1]:
↦    1,2,3
↦ [2]:
↦    [3]:
↦       1
↦ [3]:
↦    1
  string(l); // notice that intvec elements are not distinguishable
↦ 1,2,3,1,1
```

See Section 3.5.5 [Type conversion and casting], page 45; Section 5.1.117 [print], page 238; Section 4.20 [string], page 128.

4.20.4 string operations

`+` concatenation

`<=`, `>=`, `==`, `<>`
: comparison (lexicographical with respect to the ASCII encoding)

string_expression `[` int_expression `]`
: is a character of the string; the index 1 gives the first character.

string_expression `[` int_expression`,` int_expression `]`
: is a substring, where the first argument is the start index and the second is the length of the substring, filled up with blanks if the length exceeds the total size of the string

string_expression [intvec_expression]
is a expression list of characters from the string

Example:

```
  string s="abcde";
  s[2];
↦ b
  s[3,2];
↦ cd
  ">>"+s[1,10]+"<<";
↦ >>abcde     <<
  s[2]="BC"; s;
↦ aBcde
  intvec v=1,3,5;
  s=s[v]; s;
↦ ace
  s="654321"; s=s[3..5]; s;
↦ 432
```

4.20.5 string related functions

`charstr` description of the coefficient field of a ring (see Section 5.1.7 [charstr], page 160)

`execute` executing string as command (see Section 5.1.31 [execute], page 175)

`find` position of a substring in a string (see Section 5.1.41 [find], page 182)

`names` list of strings of all user-defined variable names (see Section 5.1.100 [names], page 225)

`nameof` name of an object (see Section 5.1.99 [nameof], page 225)

`option` lists all defined options (see Section 5.1.108 [option], page 230)

`ordstr` monomial ordering of a ring (see Section 5.1.110 [ordstr], page 235)

`parstr` names of all ring parameters or the name of the n-th ring parameter (see Section 5.1.113 [parstr], page 236)

`read` read a file (see Section 5.1.126 [read], page 245)

`size` length of a string (see Section 5.1.140 [size], page 260)

`sprintf` string formatting (see [sprintf], page 721)

`typeof` type of an object (see Section 5.1.156 [typeof], page 277)

`varstr` names of all ring variables or the name of the n-th ring variable (see Section 5.1.162 [varstr], page 280)

4.21 vector

Vectors are elements of a free module over the basering with basis `gen(1)`, `gen(2)`, ... Like polynomials they can only be defined or accessed with respect to the basering. Each vector belongs to a free module of rank equal to the biggest index of a generator with non-zero coefficient. Since generators with zero coefficients need not be written any vector may be considered also as an element of a free module of higher rank. (E.g., if `f` and `g` are polynomials then `f*gen(1)+g*gen(3)+gen(4)` may also be written as `[f,0,g,1]` or as `[f,0,g,1,0]`.) Note that the elements of a vector have to be surrounded by square brackets (`[` , `]`) (cf. Section B.1 [Representation of mathematical objects], page 695).

4.21.1 vector declarations

Syntax: `vector` name = vector_expression ;

Purpose: defines a vector of polynomials (an element of a free module).

Default: [0]

Example:

```
ring r=0,(x,y,z),(c,dp);
poly s1 = x2;
poly s2 = y3;
poly s3 = z;
vector v = [s1, s2-s1, s3-s1]+ s1*gen(5);
// v is a vector in the free module of rank 5
v;
↦ [x2,y3-x2,-x2+z,0,x2]
```

4.21.2 vector expressions

A vector expression is:

1. an identifier of type vector
2. a function returning vector
3. a polynomial expression (via the canonical embedding `p` ↦ `p*gen(1)`)
4. vector expressions combined by the arithmetic operations `+` or `-`
5. a polynomial expression and a vector expression combined by the arithmetic operation `*`
6. a type cast to vector using the brackets `[ , ]`

Example:

```
// ordering gives priority to components:
ring rr=0,(x,y,z),(c,dp);
vector v=[x2+y3,2,0,x*y]+gen(6)*x6;
v;
↦ [y3+x2,2,0,xy,0,x6]
vector w=[z3-x,3y];
v-w;
↦ [y3-z3+x2+x,-3y+2,0,xy,0,x6]
v*(z+x);
↦ [xy3+y3z+x3+x2z,2x+2z,0,x2y+xyz,0,x7+x6z]
// ordering gives priority to monomials:
// this results in a different output
ring r=0,(x,y,z),(dp,c);
imap(rr,v);
↦ x6*gen(6)+y3*gen(1)+x2*gen(1)+xy*gen(4)+2*gen(2)
```

See Section 3.5.5 [Type conversion and casting], page 45; Section 4.19 [ring], page 125.

4.21.3 vector operations

`+` addition

`-` negation or subtraction

`/` division by a monomial, not divisible terms yield 0

`<, <=, >, >=, ==, <>`
comparators (considering leading terms w.r.t. monomial ordering)

vector_expression `[` int_expressions `]`
is a vector entry; the index 1 gives the first entry.

Example:

```
  ring R=0,(x,y),(c,dp);
  [x,y]-[1,x];
↦ [x-1,-x+y]
  [1,2,x,4][3];
↦ x
```

4.21.4 vector related functions

`cleardenom`
quotient of a vector by its content (see Section 5.1.9 [cleardenom], page 161)

`coeffs` matrix of coefficients (see Section 5.1.12 [coeffs], page 162)

`deg` degree (see Section 5.1.18 [deg], page 167)

`diff` partial derivative (see Section 5.1.23 [diff], page 169)

`gen` i-th generator (see Section 5.1.49 [gen], page 187)

`homog` homogenization (see Section 5.1.55 [homog], page 194)

`jet` k-jet: monomials of degree at most k (see Section 5.1.66 [jet], page 202)

`lead` leading term (see Section 5.1.73 [lead], page 206)

`leadcoef` leading coefficient (see Section 5.1.74 [leadcoef], page 207)

`leadexp` the exponent vector of the leading monomial (see Section 5.1.75 [leadexp], page 207)

`leadmonom`
leading monomial (see Section 5.1.76 [leadmonom], page 208)

`nrows` number of rows (see Section 5.1.104 [nrows], page 229)

`ord` degree of the leading monomial (see Section 5.1.109 [ord], page 235)

`reduce` normal form with respect to a standard base (see Section 5.1.127 [reduce], page 246)

`simplify` normalize a vector (see Section 5.1.139 [simplify], page 258)

`size` number of monomials (see Section 5.1.140 [size], page 260)

`subst` substitute a ring variable (see Section 5.1.150 [subst], page 270)

4.22 User defined types

User defined types are (non-empty) lists with a fixed size whose element can be accessed by names (and not indices). These elements have a predefined type (which can also be a user defined type). If these elements depend on a ring they can only be accessed if their base ring ist the current base ring. In contrast to usual lists the elements of a user defined type may belong to different rings.

4.22.1 Definition of a user defined type

Syntax: `newstruct(` name `,` string_expression `);`
`newstruct(` name `,` name `,` string_expression `);`

Purpose: defines a new type with elements given by the last argument (string_expression). The name of the new type is the first argument (of type string) and must be longer than one character.
The second name (of type string) is an already defined type which should be extended by the new type.
The last argument (of type string) must be an comma separated list of a type followed by a name. If there are duplicate member names, the last one wins.
(User defined) member names are restricted to alphanumeric charecters and must start with a letter.

Operations:
the only operations of user defined types are:

assignment (between objects of the same or extended type)

`typeof`

`string` and printing

operator `.` to access the elements

Example:

```
  newstruct("nt","int a,poly b,string c");
  nt A;
  nt B;
  A.a=3;
  A.c=string(A.a);
  B=A;
  newstruct("t2","nt","string c");
  t2 C; C.c="t2-c";
  A=C;
  typeof(A);
↦ t2
  A;
↦ c=t2-c
↦ c=
↦ b=??
↦ a=0
  // a motivating example ------------------------------------------
  newstruct("IDEAL","ideal I,proc prettyprint");
  newstruct("HOMOGENEOUS_IDEAL","IDEAL","intvec weights,proc prettyprint");
  proc IDEAL_pretty_print(IDEAL I)
  {
    "ideal generated by";
    I.I;
  }
  proc H_IDEAL_pretty_print(HOMOGENEOUS_IDEAL I)
  {
    "homogeneous ideal generated by";
    I.I;
    "with weights";
```

```
    I.weights;
  }
  proc p_print(IDEAL I) { I.prettyprint(I); }
  ring r;
  IDEAL I;
  I.I=ideal(x+y2,z);
  I.prettyprint=IDEAL_pretty_print;
  HOMOGENEOUS_IDEAL H;
  H.I=ideal(x,y,z);
  H.prettyprint=H_IDEAL_pretty_print;
  H.weights=intvec(1,1,1);
  p_print(I);
↦ ideal generated by
↦ _[1]=y2+x
↦ _[2]=z
  p_print(H);
↦ homogeneous ideal generated by
↦ _[1]=x
↦ _[2]=y
↦ _[3]=z
↦ with weights
↦ 1,1,1
```

4.22.2 Declaration of objects of a user defined type

Example:

```
  newstruct("nt","int a,poly b,string c");
  nt A;
  // as long as there is no value assigned to A.b, no ring is needed
  nt B=A;
```

4.22.3 Access to elements of a user defined type

Access to elements of a user defined type via `.`: <object>.<element_name>. The <element_names> are from the definition of the type. Additional, all (potentially) ring dependent elements have an additional entry `r_`<element_name> for the corresponding ring.

Example:

```
  newstruct("nt","int a,poly b,string c");
  nt A;
  3+A.a;
↦ 3
  A.c="example string";
  ring r;
  A.b=poly(1); // assignment: expression must be of the given type
  A;
↦ c=example string
↦ b=1
↦ a=0
  A.r_b;
↦ // coefficients: ZZ/32003
↦ // number of vars : 3
↦ //        block   1 : ordering dp
```

```
↦ //                  : names    x y z
↦ //        block   2 : ordering C
```

4.22.4 Commands for user defined types

User defined types are normal data types (which do not belong to a ring, even if they have ring dependent parts), so they can be passed as argument to procedures, and recieved as result from procedures.

In order to apply kernel commands to these types (like `string`, `+`), provide a usual procedure (say `proc p..`) for that task and install it via `system("install",` user_type , kernel_command `,p,` number_of_args `);`. The user_type and kernel_command have to be given as strings. For kernel_command having a variable number of arguments (internal `CMD_M`) use 4 independ of the number of really supplied arguments.

List of available kernel commands and the required number_of_args, some accept several variants and appear therefor at several places:

inplace binary operands: `+`,`-`,`*`,`/`,`div`,`%`,`&`,`|`, `[`, number_of_args:2

unary functions: `attrib`, `bareiss`, `betti`, `char`, `char_series`, `charstr`, `cleardenom`, `close`, `convhull`, `defined`, `deg`, `degree`, `denominator`, `det`, `dim`, `dump`, `ERROR`, `envelope`, `execute`, `facstd`, `factorize`, `finduni`, `gen`, `getdump`, `hilb`, `impart`, `indepSet`, `interred`, `jacob`, `janet`, `kbase`, `killattrib`, `lead`, `leadcoef`, `leadexp`, `leadmonom`, `load`, `ludecomp`, `maxideal`, `memory`, `minbase`, `minres`, `monitor`, `monomial`, `mult`, `mstd`, `nameof`, `ncols`, `npars`, `nrows`, `numerator`, `nvars`, `open`, `opposite`, `ord`, `ordstr`, `par`, `pardeg`, `parstr`, `preimage`, `prime`, `primefactors`, `prune`, `qhweight`, `rank`, `read`, `regularity`, `repart`, `ringlist`, `rvar`, `sba`, `size`, `slimgb`, `sortvec`, `sqrfree`, `syz`, `trace`, `transpose`, `twostd`, `typeof`, `univariate`, `var`, `variables`, `varstr`, `vdim`, `waitfirst`, `waitall`, `weight`

functions with 2 arguments: `attrib`, `betti`, `bracket`, `chinrem`, `coeffs`, `contract`, `deg`, `delete`, `diff`, `dim`, `extgcd`, `eliminate`, `exportto`, `facstd`, `factorize`, `farey`, `fetch`, `fglm`, `fglmquot`, `find`, `fres`, `frwalk`, `gcd`, `hilb`, `homog`, `hres`, `imap`, `importfrom`, `indepSet`, `insert`, `interpolation`, `janet`, `kbase`, `kernel`, `killattrib`, `koszul`, `lift`, `liftstd`, `load`, `lres`, `modulo`, `mpresmat`, `mres`, `newstruct`, `nc_algebra`, `nres`, `oppose`, `parstr`, `primefactors`, `quotient`, `random`, `rank`, `read`, `sba`, `simplify`, `sqrfree`, `sres`, `varstr`, `waitfirst`, `waitall`, `wedge`

functions with 3 arguments: `attrib`, `bareiss`, `coeffs`, `eliminate`, `find`, `fres`, `frwalk`, `hilb`, `homog`, `insert`, `koszul`, `laguerre`, `lift`, `liftstd`, `newstruct`, `preimage`, `random`, `resultant`, `sba`, `vandermonde`

functions with variable number of arguments arguments (number_of_args:4): `breakpoint`, `coef`, `dbprint`, `division`, `factmodd`, `intersect`, `jet`, `luinverse`, `lusolve`, `minor`, `names`, `option`, `qrds`, `reduce`, `reservedName`, `simplex`, `status`, `std`, `subst`, `system`, `test`, `uressolve`, `write`

Example:

```
  newstruct("nt","int a,poly b,string c");
  nt A;
  A;
↦ c=
↦ b=??
↦ a=0
  ring r;
```

```
// a pretty print routine for nt:
proc pretty_print(nt A)
{
  "nt with string c:"+A.c+" and poly:"+string(A.b);
}
system("install","nt","print",pretty_print,1); // default printing uses print
A;
↦ nt with string c: and poly:0
↦
// a custem add for nt:
proc nt_add(nt A,nt B)
{
  nt C;
  C.a=A.a+B.a; C.b=A.b+B.b; C.c=A.c+B.c;
  return(C);
}
system("install","nt","+",nt_add,2);
A.b=x;
nt B; B.c="B"; B.b=y;
A+B;
↦ nt with string c:B and poly:x+y
↦
```

4.22.5 Assignments for user defined types

By default, only objects of the same (user defined) type can be assigned, there is no automatic type conversion as for the kernel data types.

But the operator `=` can be overidden in oder to write custom constructors (the custom contructor does not apply to assignments of the same type): via `system("install",` user_type `,"=" ,p,1);`. The user_type has to be given as a string.

Example:

```
newstruct("wrapping","poly p");
proc wrap(poly p)
{
  wrapping w; w.p = p;
  return  (w);
}
system("install", "wrapping", "=", wrap, 1);
ring r = 0,x,dp;
wrapping w = x+1;
w;
↦ p=x+1
w = int(1); // via conversion int->poly
w;
↦ p=1
w=number(2); // via conversion number->poly
w;
↦ p=2
```

The user defined procedure for `=` provides also generic type conversions: `hh A=hh(b);` is equivalent to `hh tmp=b; hh A=tmp; kill tmp;`.

4.23 cone

In order to use convex objects in Singular, Singular uses gfanlib, a C++ library for convex geometry by Anders N. Jensen. Please check the readme file for installation instructions. The library Section D.13.1 [gfan_lib], page 811 provides this C++ library and an interface to it. Some functions require `polymake`: Section D.13.2 [polymake_lib], page 814 provides an interface to it.

In the finite dimensional real vector space R^n, a convex rational polyhedral cone, in short "cone", is the convex set generated by finitely many half-lines, which in turn are generated by rational, and hence integer, points. It may or may not contain whole subspace of R^n (e.g. entire lines). The biggest subspace contained in a cone is called "lineality space". Modulo its lineality space, each cone is generated by a distinct minimal set of half lines, which are referred to as "rays". Alternatively, a cone can be represented as a set of points satisfying a system homogeneous rational, and hence integer, linear inequalities and equations. These two characterizations of cones are the two main ways of defining cones in Singular (see [coneViaPoints], page 811, see [coneViaInequalities], page 811).

```
LIB"gfan.lib";
cone c;                                  // ambient dim 0, no equations,
                                         // no inequalities
cone c = 17;                             // ambient dim 17, no equations,
                                         // no inequalities
```

See Section D.13.1 [gfan_lib], page 811.

4.24 fan

In order to use convex objects in Singular, Singular uses gfanlib, a C++ library for convex geometry by Anders N. Jensen. Please check the readme file for installation instructions. The library Section D.13.1 [gfan_lib], page 811 provides this C++ library and an interface to it. Some functions require `polymake`: Section D.13.2 [polymake_lib], page 814 provides an interface to it.

A polyhedral fan is a collection of cones closed under taking intersections and faces.

```
fan f;                                   // ambient dim 0, no cones, lineality
                                         // space = ambient space
fan f = emptyFan(int n);                 // ambient dim = n, no cones,
                                         // symmetry subgroup = <e>
fan f = n;                               // synonymous convenience method for
                                         // 'fan f = emptyFan(n)'
fan f = emptyFan(
               optional intmat G);       // symmetry group generated by rows of
                                         // G; ambient dim = number of columns
                                         // of G
          // Here a row r = r_1, ...r_n represents the permutation
          // which takes i to r_i. SINGULAR will check for the validity
          // of the provided rows, i.e. whether {r_1,...,r_n} = {1,...,n}
fan f = fullFan(int n);                  // ambient dim = n, the fan consists of
                                         // one cone which equals the ambient
                                         // space, symmetry subgroup = <e>
fan f = fullFan(
               optional intmat G);       // symmetry group generated by rows of
                                         // G; ambient dim = number of columns
                                         // of G; the fan consists of one cone
                                         // which equals the ambient space
```

See also Section D.13.1 [gfan_lib], page 811.

4.25 polytope

In order to use convex objects in Singular, Singular uses gfanlib, a C++ library for convex geometry by Anders N. Jensen. Please check the readme file for installation instructions. The library Section D.13.1 [gfan_lib], page 811 provides this C++ library and an interface to it. Some functions require `polymake`: Section D.13.2 [polymake_lib], page 814 provides an interface to it.

In R^n, a rational convex polytope (in short "polytope") is a set of points satisfying a system rational, and hence integer, linear inequalities and equations. It is internally realised as a cone in R^(n+1) intersected with the hyperplane, in which the first coordinate equals 1.

See Section D.13.1 [gfan_lib], page 811.

4.26 pyobject

The pyobject is a black box data type in Singular for handling objects from the programming language `python`. It needs the `python` support of Singular to be installed.

Together with some basic operations and funtions, pyobject instances access `python` functionality from within Singular and store the results for re-use:

Note that this feature is automatically loaded on demand when initializing an object of type `pyobject`. For accessing `pyobject`-related functions before using any `python` object, please type `LIB("pyobject.so");` at the Singular prompt.

```
pyobject pystr = "Hello";
pyobject pyint = 2;
string singstr = string(pystr + " World!");
singstr;
↦ 'Hello World!'
pystr + pyint;  // Error: not possible
↦    ? pyobject error occurred
↦    ? cannot concatenate 'str' and 'int' objects
↦    ? error occurred in or before ./examples/pyobject.sing line 5: `pystr \
   + pyint;  // Error: not possible`
pystr * pyint;  // But this is allowed,
↦ 'HelloHello'
pystr * 3;      // as well as this;
↦ 'HelloHelloHello'

python_run("def newfunc(*args): return list(args)");  // syncs contexts!
newfunc(1, 2, 3);          // newfunc also knowd to SINGULAR
↦ [1, 2, 3]

def pylst = python_eval("[3, 7, 1]");
proc(attrib(pylst, "sort"))(); // Access python member routines as attributes
pylst.sort();             // <- equivalent short-notation
pylst."sort"();           // <- alternative short-notation
pylst;
↦ [1, 3, 7]

python_import("os");       // Gets stuff from python module 'os'
name;                      // The identifier of the operating system
↦ 'posix'
```

4.26.1 pyobject declarations

Syntax: pyobject name = pyobject_expression ;

Purpose: defines a `python` object.

Default: `None`

Example:

```
  pyobject empty;
  empty;
↦ None

  pyobject pystr = "Hello World!";
  pyobject pyone = 17;
  pyobject pylst = list(pystr, pyone);
  pylst;
↦ ['Hello World!', 17]
```

4.26.2 pyobject expressions

A pyobject expression is (optional parts in square brackets):

1. an identifier of type pyobject
2. a function returning pyobject
3. pyobject expressions combined by the arithmetic operations `+`, `-`, `*`, `/`, or `^`, and the member-of operators `.` and `::`
4. an list expression with elements made of pyobject expressions (see Section 3.5.5 [Type conversion and casting], page 45)
5. an string expression (see Section 3.5.5 [Type conversion and casting], page 45)
6. an int expression (see Section 3.5.5 [Type conversion and casting], page 45)

Example:

```
  pyobject pystr = "python string ";
  pystr;
↦ 'python string '
  pyobject pyint = 2;
  pyint;
↦ 2
  pyobject pylst = list(pystr, pyint);
  pylst;
↦ ['python string ', 2]
  pyint + pyint;
↦ 4
  pyint * pyint;
↦ 4
  pystr + pystr;
↦ 'python string python string '
  pystr * pyint;
↦ 'python string python string '
  python_eval("17 + 4");
↦ 21
  typeof(_);
↦ pyobject
```

4.26.3 pyobject operations

`+` addition

`-` negation or subtraction

`*` multiplication

`/` division

`^`, `**` power by a positive integer

`<`, `<=`, `>`, `>=`, `==`, `<>`
comparators (considering leading monomials w.r.t. monomial ordering)

pyobject_expression `[` int_expression `]`
get the item from the pyobject by index

pyobject_expression `(` pyobject_expression_sequence `)`
call the pyobject with a sequence of python arguments (the latter may be empty)

pyobject_expression `.` (string_expression `|` name),
pyobject_expression `::` (string_expression `|` name) get attribute (class member) of a python object

Example:

```
pyobject two = 2;
pyobject three = 3;

two + three;
↦ 5
two - three;
↦ -1
two * three;
↦ 6
two / three;
↦ 0
two ^ three;
↦ 8
two ** three;
↦ 8

three < two;
↦ 0
two < three;
↦ 1
three <= two;
↦ 0
two <= three;
↦ 1
two == three;
↦ 0
two == two;
↦ 1
three > two;
↦ 1
```

```
two > three;
↦ 0
three >= two;
↦ 1
two >= three;
↦ 0
two != two;
↦ 0
two != three;
↦ 1

pyobject pystr = "Hello";
pystr + " World!";
↦ 'Hello World!'
pystr * 3;
↦ 'HelloHelloHello'
pystr[1];
↦ 'e'

python_run("def newfunc(*args): return list(args)");
newfunc();
↦ []
newfunc(two, three);
↦ [2, 3]

newfunc."__class__";
↦ <type 'function'>
newfunc::"__class__";
↦ <type 'function'>
newfunc.func_name;
↦ 'newfunc'
newfunc::func_name;
↦ 'newfunc'
```

4.26.4 pyobject related functions

`attrib` list, get and set attributes (class members) of a pyobject (see Section 5.1.2 [attrib], page 154)

Example:

```
  pyobject pystr = "Kublai Khan";

  // Additional functionality through attrib
  attrib(pystr, "__doc__");
↦ "str(object='') -> string\n\nReturn a nice string representation of the o\
   bject.\nIf the argument is a string, the return value is the same object.\
   "
  proc(attrib(pystr, "count"))("K");
↦ 2

  pystr."__doc__";           // <- Short notations
```

```
↦ "str(object='') -> string\n\nReturn a nice string representation of the o\
   bject.\nIf the argument is a string, the return value is the same object.\
   "
  pystr.count("a");          // Even shorter (if attribute's name is valid and unused)
↦ 2

  python_run("def func(): return 17");
  attrib(func);
↦ ['__call__', '__class__', '__closure__', '__code__', '__defaults__', '__d\
   elattr__', '__dict__', '__doc__', '__format__', '__get__', '__getattribut\
   e__', '__globals__', '__hash__', '__init__', '__module__', '__name__', '_\
   _new__', '__reduce__', '__reduce_ex__', '__repr__', '__setattr__', '__siz\
   eof__', '__str__', '__subclasshook__', 'func_closure', 'func_code', 'func\
   _defaults', 'func_dict', 'func_doc', 'func_globals', 'func_name']
  attrib(func, "func_name");
↦ 'func'
  attrib(func, "func_name", "byAnyOtherName");
  attrib(func, "func_name");
↦ 'byAnyOtherName'
```

`killattrib`
deletes an attribute from a pyobject (see Section 5.1.70 [killattrib], page 204)

Example:

```
LIB("pyobject.so");
python_run("def new_pyobj(): pass");
attrib(new_pyobj, "new_attr", "something");
attrib(new_pyobj, "new_attr");
↦ 'something'
attrib(new_pyobj);
↦ ['__call__', '__class__', '__closure__', '__code__', '__defaults__', '__d\
   elattr__', '__dict__', '__doc__', '__format__', '__get__', '__getattribut\
   e__', '__globals__', '__hash__', '__init__', '__module__', '__name__', '_\
   _new__', '__reduce__', '__reduce_ex__', '__repr__', '__setattr__', '__siz\
   eof__', '__str__', '__subclasshook__', 'func_closure', 'func_code', 'func\
   _defaults', 'func_dict', 'func_doc', 'func_globals', 'func_name', 'new_at\
   tr']

killattrib(new_pyobj, "new_attr");
attrib(new_pyobj);
↦ ['__call__', '__class__', '__closure__', '__code__', '__defaults__', '__d\
   elattr__', '__dict__', '__doc__', '__format__', '__get__', '__getattribut\
   e__', '__globals__', '__hash__', '__init__', '__module__', '__name__', '_\
   _new__', '__reduce__', '__reduce_ex__', '__repr__', '__setattr__', '__siz\
   eof__', '__str__', '__subclasshook__', 'func_closure', 'func_code', 'func\
   _defaults', 'func_dict', 'func_doc', 'func_globals', 'func_name']
```

`python_run`
execute string-given `python` commands and import new symbols from `python` to Singular's context (see Section 4.26.7 [python_run], page 144).

`python_eval`
evaluate a string-given `python` expression and return the result to Singular (see Section 4.26.5 [python_eval], page 144).

`python_import`
: import **python** module into SINGULAR's context (see Section 4.26.6 [python_import], page 144)

4.26.5 python_eval

Syntax: `python_eval` (string_expression)

Type: pyobject

Purpose: Evaluates a python expression (given as a string) and returns the result as pyobject.

Example:

```
  LIB("pyobject.so");
  python_eval("17 + 4");
↦ 21
  typeof(_);
↦ pyobject
  list ll = python_eval("range(10)");
```

4.26.6 python_import

Syntax: `python_import` (string_expression)

Type: pyobject

Purpose: Imports python module (given as a string) in the SINGULAR context.

Example:

```
  LIB("pyobject.so");
  python_import("os");
  name;                         // e. g. 'posix'
↦ 'posix'
  sep;                          // pathname separator
↦ '/'
  linesep;                      // end of line marker
↦ '\n'
```

4.26.7 python_run

Syntax: `python_run` (string_expression)

Type: none

Purpose: Executes python commands (given as a string) in `python` context and syncs the contexts afterwards.

Example:

```
  LIB("pyobject.so");
  python_run("def newfunc(*args): return list(args)");
  newfunc(1, 2, 3);          // newfunc also known to SINGULAR now
↦ [1, 2, 3]

  python_run("import os");
  os;
↦ <module 'os' from '/usr/lib/python2.7/os.pyc'>
  attrib(os, "name");
↦ 'posix'
```

4.27 reference and shared (experimental)

The black box data types `reference` and `shared` in SINGULAR allow for concurrently accessing SINGULAR object data. Copying such object will only add an additional handle which allows you to define multiple identifers for the same object instance.

Both experimental features are hidden by default, please activate them by typing `system("reference");` or `system("shared");`, respectively, at the SINGULAR prompt.

You must initialize a `reference` using a named identifier or a subexpression of the latter. The resulting object can be stored to gain read and write access from sophisticated data structures.

```
system("reference"); system("shared");
int i = 17;
reference ref = i;

ref;
↦ 17
↦
ref = 19;
ref;
↦ 19
↦
i;              // original handle changed!
↦ 19

kill ref;
i;              // 'i' stays alive
↦ 19

reference uninitialized;
uninitialized;   // not initialized
↦ <unassigned reference or shared memory>
// error: not a named identifier:
uninitialized = 17;
↦    ? Can only take reference from identifier
↦    ? error occurred in or before ./examples/reference_and_shared__experim\
   ental_.sing line 16: `uninitialized = 17;`

// but subexpressions of named identifiers will do
list ll = list(3,4,5);
reference ref = ll[2];
ref;
↦ 4
↦
ref = 12;
ref;
↦ 12
↦
ll;
↦ [1]:
↦    3
↦ [2]:
↦    12
↦ [3]:
```

```
↦    5
```

In contrast, the type `shared` can be used to avoid the initial identifier definition. Each copy has equal rights for manipulating the data.

```
system("reference"); system("shared");
shared ll= list(2,3);

ll[1];
↦ 2
↦
ll[1]= 17;
ll;
↦ [1]:
↦    17
↦ [2]:
↦    3
↦
```

In most cases the value look-up is done automatically, but sometimes you have to disambiguate the input.

```
system("reference"); system("shared");
int i = 0;
reference ref = i;
shared sh = 12;

ref + sh;    // automated 'dereferencing'
↦ 12
ref + 4;
↦ 4
4 + sh;
↦ 16

list ll = list(ref, ref, ref, ref, ref, ref, ref);
string(ll);
↦ 0,0,0,0,0,0,0
ref = 1;
string(ll);   // all one now
↦ 1,1,1,1,1,1,1

ll[3] = 0;
string(ll);   // only third element changed
↦ 1,1,0,1,1,1,1

reference(ll[1]) = 9;
string(ll);   // all others changed
↦ 9,9,0,9,9,9,9

def(ll[1]) = 11;  // alternative (generic) syntax
string(ll);
↦ 11,11,0,11,11,11,11
```

The previous example had shown that `reference` and `shared` objects can store highly structured without duplicating data all over again. As an additional feature, you can use `reference` objects for implementing procedures having side-effects.

```
system("reference"); system("shared");
list changeme;
changeme;
↦ empty list

proc setfirst(reference ll, def arg) {  ll[1] = arg; }

setfirst(changeme, 17);
changeme;
↦ [1]:
↦    17
```

If you do not need write-access to `proc` parameters, your code will usually perform better using the `alias` statement in the parameter list, see Section 4.17 [proc], page 122.

4.27.1 reference declarations

Syntax: reference name = identifier ;

Purpose: defines a `reference` object.

Default: `None`

Example:

```
system("reference"); system("shared");
  reference empty;
  empty;
↦ <unassigned reference or shared memory>

  string str = "Hello World!";
  reference ref = str;
  ref;
↦ Hello World!
↦
  ref = 17;    // cannot change type of 'i'
↦    ? `string`(str) = `int` is not supported
↦    ? expected `string` = `string`
↦    ? error occurred in or before ./examples/reference_declarations.sing l\
   ine 8: `  ref = 17;    // cannot change type of 'i'`
  list ll= list(4, 5, 6);
  reference lref = ll[2];
  lref;
↦ 5
↦
  lref = str;  // change list element
  ll;
↦ [1]:
↦    4
↦ [2]:
↦    Hello World!
↦ [3]:
↦    6
```

4.27.2 reference expressions

A reference expression:

1. any identifier
2. any subexpression of an identifier
3. an object of type `reference` (result will reference the original identifier, too)

Example:

```
system("reference"); system("shared");
  int i = 17;
  reference ref = i;  // new reference
  ref;
↦ 17
↦
  reference second = ref;
  second;
↦ 17
↦
  second = 9;       // also tied to 'i'
  i;
↦ 9
  typeof(ref);
↦ reference

  list ll = list(1, 2, 3);
  reference lref = ll[1];
  lref;
↦ 1
↦
  lref = 12;
  ll;
↦ [1]:
↦    12
↦ [2]:
↦    2
↦ [3]:
↦    3
```

4.27.3 shared declarations

Syntax: shared name = expression ;

Purpose: defines a `shared` object.

Default: `None`

Example:

```
system("reference"); system("shared");
  shared empty;
  empty;
↦ '_'
↦

  shared str = "Hello World!";
  str;
↦ Hello World!
```

```
↦
  shared ll= list(4, 5, 6);
  ll;
↦ [1]:
↦    4
↦ [2]:
↦    5
↦ [3]:
↦    6
↦
  ll[2] = str;  // change list element
  ll;
↦ [1]:
↦    4
↦ [2]:
↦    Hello World!
↦ [3]:
↦    6
↦
```

4.27.4 shared expressions

shared expression:

1. any expression
2. an object of type `shared` (result will reference the same data)

Example:

```
system("reference"); system("shared");
  shared sh = 17;  // new shared
  shared second = sh;
  second;
↦ 17
↦
  second = 9;       // also tied to 'sh'
  sh;
↦ 9
↦
  typeof(sh);
↦ shared

  shared ll = list(1, 2, 3);
  shared lref = ll[1];
  lref;
↦ 1
↦
  lref = 12;
  ll;
↦ [1]:
↦    12
↦ [2]:
↦    2
↦ [3]:
```

```
↦    3
↦
```

4.27.5 reference and shared operations

All operations of the underlying objects are forwarded by `reference` and `shared` objects. THis kind of dereferencing is done automatically in most cases:

Example:

```
system("reference"); system("shared");
int i = 2;
reference two = i;
shared three = 3;

two * three;
↦ 6
two ^ three;
↦ 8
two ** three;
↦ 8

two + two;
↦ 4
two - two;
↦ 0

ring r = 0, (x,y,z), dp;
poly p = x + y + z;
reference ref = p;
shared zvar =z;
subst(ref, x,1, y,2, zvar,3);
↦ 6
```

In some cases `reference`s have to be dereferenced explicitely. For instance, this is the case for n-ary function calls not starting with a `reference` or `shared` object. You can use the `link` operator or a type cast to work around this. In contrast, some constructs like left-hand subexpressions prematurely evaluate. You can avoid this by using the `def` operator or by explicitly type casting to `reference`.

```
system("reference"); system("shared");
ring r = 0, (x,y,z), dp;
poly p = x + y + z;
shared xsh = x;
subst(p, xsh,1, y,2, z,3);        // fails
↦    ? subst(`poly`,`shared`,`int`) failed
↦    ? expected subst(`poly`,`poly`,`poly`)
↦    ? expected subst(`matrix`,`poly`,`int`)
↦    ? error occurred in or before ./examples/reference_and_shared_operatio\
   ns_1.sing line 5: `subst(p, xsh,1, y,2, z,3);        // fails`
subst(p, poly(xsh),1, y,2, z,3);  // good
↦ 6
subst(p, link(xsh),1, y,2, z,3);  // fine
↦ 6
```

```
list ll = list(xsh, xsh, xsh);
ll[1] = y;      // replaced only first entry
ll;
↦ [1]:
↦    y
↦ [2]:
↦    x
↦
↦ [3]:
↦    x
↦
shared(ll[2]) = z;    // replaces the others
ll;
↦ [1]:
↦    y
↦ [2]:
↦    z
↦
↦ [3]:
↦    z
↦
def(ll[2]) = x;       // generic alternative
ll;
↦ [1]:
↦    y
↦ [2]:
↦    x
↦
↦ [3]:
↦    x
↦
```

In particular, explicit dereferencing is useful to distinguish between typecasting and nested constructings.

```
system("reference"); system("shared");
shared shl = list(1);
shl;
↦ [1]:
↦    1
↦
list(shl);  // wraps 'shl' by a list
↦ [1]:
↦    [1]:
↦    1
↦
link(shl);  // extract the list in 'shl'
↦ [1]:
↦    1
```

4.27.6 reference and shared related functions

def explicitly type casts to **reference** or **shared**, respectively. (Note: For the **def** declaration, see Section 4.4 [def], page 78.)

Example:

```
system("reference"); system("shared");
int i =1;
reference ref = i;
shared sh = 17;
list ll = list(ref, ref, ref, sh, sh);
ll[1] = 2;      // replace only one entry
ll;
↦ [1]:
↦    2
↦ [2]:
↦    1
↦
↦ [3]:
↦    1
↦
↦ [4]:
↦    17
↦
↦ [5]:
↦    17
↦
def(ll[2]) = 3;      // change the others
ll;
↦ [1]:
↦    2
↦ [2]:
↦    3
↦
↦ [3]:
↦    3
↦
↦ [4]:
↦    17
↦
↦ [5]:
↦    17
↦
def(ll[4]) = 19;     // same here
ll;
↦ [1]:
↦    2
↦ [2]:
↦    3
↦
↦ [3]:
↦    3
↦
↦ [4]:
↦    19
↦
↦ [5]:
```

```
↦    19
↦
```

`link` explicitly dereference a `reference` or `shared` object. (Note: For the `link` declaration, see Section 4.9 [link], page 95.)

Example:

```
system("reference"); system("shared");
ring r = 0, (x,y,z), dp;
poly p = x + y + z;
def x_=x;
reference xref=x_;
xref;
↦ x
↦
subst(p, xref,1, y,2, z,3);        // fails
↦    ? subst(`poly`,`reference`,`int`) failed
↦    ? expected subst(`poly`,`poly`,`poly`)
↦    ? expected subst(`matrix`,`poly`,`int`)
↦    ? error occurred in or before ./examples/reference_and_shared_related_\
   functions_1.sing line 7: `subst(p, xref,1, y,2, z,3);        // fails`
subst(p, link(xref),1, y,2, z,3);  // fine
↦ 6
```

`system` The `reference` and `shared` objects overload the `system` command to gain extended features, see `system(ref, "help")` for more details. (Note: For the general `system` command, see Section 5.1.151 [system], page 270.)

Example:

```
system("reference"); system("shared");
shared sh;
system(sh, "help");
↦ system(<ref>, ...): extended functionality for reference/shared data <ref\
   >
↦   system(<ref>, count)         - number of references pointing to <ref>
↦   system(<ref>, enumerate)     - unique number for identifying <ref>
↦   system(<ref>, undefined)     - checks whether <ref> had been assigned
↦   system(<ref>, "help")        - prints this information message
↦   system(<ref>, "typeof")      - actual type referenced by <ref>
↦   system(<ref1>, same, <ref2>) - tests for identic reference objects
```

5 Functions and system variables

5.1 Functions

This section gives a complete reference of all functions, commands and special variables of the SINGULAR kernel (i.e., all built-in commands). See Section D.1 [standard_lib], page 721, for those functions from the `standard.lib` (this library is automatically loaded at start-up time) which extend the functionality of the kernel and are written in the SINGULAR programming language.

The general syntax of a function is

[target =] function_name (<arguments>);

If no target is specified, the result is printed. In some cases (e.g., `export`, `keepring`, `setring`, `type`) the brackets are optional. For the commands `kill`, `help`, `break`, `quit`, `exit` and `LIB` no brackets are allowed.

5.1.1 align

Syntax: `align` (vector_expression, int_expression)
`align` (module_expression, int_expression)

Type: type of the first argument

Purpose: maps module generators `gen(i)` to `gen(i+s)` for all i.

Example:

```
  ring r=0,(x,y,z),(c,dp);
  align([1,2,3],3);
↦ [0,0,0,1,2,3]
  align([0,0,1,2,3],-1);
↦ [0,1,2,3]
  align(freemodule(2),1);
↦ _[1]=[0,1]
↦ _[2]=[0,0,1]
```

5.1.2 attrib

Syntax: `attrib` (name)

Type: none

Purpose: displays the attribute list of the object called name.

Example:

```
  ring r=0,(x,y,z),dp;
  ideal I=std(maxideal(2));
  attrib(I);
↦ attr:isSB, type int
```

Syntax: `attrib` (name , string_expression)

Type: any

Purpose: returns the value of the attribute string_expression of the variable name. If the attribute is not defined for this variable, `attrib` returns the empty string.

Example:

```
  ring r=0,(x,y,z),dp;
  ideal I=std(maxideal(2));
  attrib(I,"isSB");
↦ 1
  // maxideal(2) is a standard basis,
  // SINGULAR does know it for maxideal:
  attrib(maxideal(2), "isSB");
↦ 1
```

Syntax: `attrib ( name, string_expression, expression )`

Type: none

Purpose: sets the attribute string_expression of the variable name to the value expression.

Example:

```
  ring r=0,(x,y,z),dp;
  ideal I=maxideal(2); // the attribute "isSB" is not set
  vdim(I);
↦ 4
  attrib(I,"isSB",0);  // the standard basis attribute is reset here
  vdim(I);
↦ // ** I is no standard basis
↦ 4
```

Remark: An attribute may be described by any string_expression. Some of these are used by the kernel of SINGULAR and referred to as reserved attributes. Non-reserved attributes may be used, however, in procedures and can considerably speed up computations.

Reserved attributes:
(cf_class, global, isSB, isHomog, rank, ring_cf, rowShift are used by the kernel, the other are used by libraries)

`cf_class` (for `ring`)
: the internal type of the coefficients (see `n_coeffType`)

`global` (for `ring`)
: 1, if the ordering is global

`isSB` (for `ideal, module`)
: the standard basis property is set by all commands computing a standard basis like `groebner`, `std`, `stdhilb` etc.; used by `lift`, `dim`, `degree`, `mult`, `hilb`, `vdim`, `kbase`

`isHomog` (for `ideal, module`)
: the weight vector for homogeneous or quasihomogeneous ideals/modules

`isCI` complete intersection property

`isCM` Cohen-Macaulay property

`maxExp` (for `ring/list` from `ringlist`)
: limit for each exponent (32767 by default)

`rank` (for `module`)
: set/get the rank of a module (see Section 5.1.104 [nrows], page 229)

`ring_cf` (for `ring`)
: the coefficients of the polynomial ring are considered to be a ring

`withSB` value of type ideal, resp. module, is std

`withHilb` value of type intvec is hilb(_,1) (see Section 5.1.54 [hilb], page 193)

`withRes` value of type list is a free resolution

`withDim` value of type int is the dimension (see Section 5.1.24 [dim], page 170)

`withMult` value of type int is the multiplicity (see Section 5.1.98 [mult], page 224)

See Section 5.1.70 [killattrib], page 204.

5.1.3 bareiss

Syntax: `bareiss (` module_expression `)`
`bareiss (` matrix_expression `)`
`bareiss (` module_expression`,` int_expression`,` int_expression `)`
`bareiss (` matrix_expression`,` int_expression`,` int_expression `)`

Type: list of module and intvec

Purpose: applies the sparse Gauss-Bareiss algorithm (see Section C.9 [References], page 719, Lee and Saunders) to a module (or with type conversion to a matrix) with an 'optimal' pivot strategy. The vectors of the module are the columns of the matrix, hence elimination takes place w.r.t. rows.
With only one parameter a complete elimination is done. Result is a list: the first entry is a module with a minimal independent set of vectors (as a matrix lower triangular), the second entry an intvec with the permutation of the rows w.r.t. the original matrix, that is, a k at position l indicates that row k was carried over to the row l.
The further parameters control the algorithm. `bareiss(M,i,j)` does not attempt to diagonalize the last i rows in the elimination procedure and stops computing when the remaining number of vectors (columns) to reduce is at most j.

Example:

```
  ring r=0,(x,y,z),(c,dp);
  module mm;
  // ** generation of the module mm **
  int d=7;
  int b=2;
  int db=d-b;
  int i;
  for(i=d;i>0;i--){ mm[i]=3*x*gen(i); }
  for(i=db;i;i--){ mm[i]=mm[i]+7*y*gen(i+b); }
  for(i=d;i>db;i--){ mm[i]=mm[i]+7*y*gen(i-db); }
  for(i=d;i>b;i--){ mm[i]=mm[i]+11*z*gen(i-b); }
  for(i=b;i;i--){ mm[i]=mm[i]+11*z*gen(i+db); }
  // ** the generating matrix of mm **
  print(mm);
↦ 3x, 0,  11z,0,  0,  7y, 0,
↦ 0,  3x, 0,  11z,0,  0,  7y,
↦ 7y, 0,  3x, 0,  11z,0,  0,
↦ 0,  7y, 0,  3x, 0,  11z,0,
↦ 0,  0,  7y, 0,  3x, 0,  11z,
↦ 11z,0,  0,  7y, 0,  3x, 0,
↦ 0,  11z,0,  0,  7y, 0,  3x
  // complete elimination
```

```
  list ss=bareiss(mm);
  print(ss[1]);
↦ 7y, 0,     0,     0,      0,        0,     0,
↦ 3x, -33xz, 0,     0,      0,        0,     0,
↦ 11z,-121z2,1331z3,0,      0,        0,     0,
↦ 0,  0,     0,     9317yz3,0,        0,     0,
↦ 0,  21xy,  _[5,3],14641z4,-43923xz4,0,     0,
↦ 0,  0,     0,     0,      65219y2z3,_[6,6],0,
↦ 0,  49y2,  _[7,3],3993xz3,_[7,5],   _[7,6],_[7,7]
  ss[2];
↦ 2,7,5,1,4,3,6
  // elimination up to 3 vectors
  ss=bareiss(mm,0,3);
  print(ss[1]);
↦ 7y, 0,     0,     0,      0,        0,        0,
↦ 3x, -33xz, 0,     0,      0,        0,        0,
↦ 11z,-121z2,1331z3,0,      0,        0,        0,
↦ 0,  0,     0,     9317yz3,0,        0,        0,
↦ 0,  0,     0,     0,      27951xyz3,102487yz4,65219y2z3,
↦ 0,  21xy,  _[6,3],14641z4,_[6,5],   _[6,6],   -43923xz4,
↦ 0,  49y2,  _[7,3],3993xz3,_[7,5],   _[7,6],   _[7,7]
  ss[2];
↦ 2,7,5,1,3,4,6
  // elimination without the last 3 rows
  ss=bareiss(mm,3,0);
  print(ss[1]);
↦ 7y, 0,   0,      0,       0,     0,     0,
↦ 0,  77yz,0,      0,       0,     0,     0,
↦ 0,  0,   231xyz, 0,       0,     0,     0,
↦ 0,  0,   0,      1617xy2z,0,     0,     0,
↦ 11z,21xy,-1331z3,14641z4, _[5,5],_[5,6],_[5,7],
↦ 0,  0,   539y2z, _[6,4],  _[6,5],_[6,6],-3773y3z,
↦ 3x, 49y2,-363xz2,3993xz3, _[7,5],_[7,6],_[7,7]
  ss[2];
↦ 2,3,4,1
```

See Section 5.1.22 [det], page 169; Section 4.12 [matrix], page 107.

5.1.4 betti

Syntax: `betti` (list_expression)
`betti` (resolution_expression)
`betti` (list_expression , int_expression)
`betti` (resolution_expression , int_expression)

Type: intmat

Purpose: with 1 argument: computes the graded Betti numbers of a minimal resolution of R^n/M, if R denotes the basering, M is a homogeneous submodule of R^n and the argument represents a resolution of R^n/M.
The entry d of the intmat at place (i,j) is the minimal number of generators in degree i+j of the j-th syzygy module (= module of relations) of R^n/M, i.e. the 0th (resp. 1st) syzygy module of R^n/M is R^n (resp. M). The argument is considered to be the result of a res/fres/sres/mres/nres/lres command. This implies that a zero is only allowed

(and counted) as a generator in the first module.
For the computation betti uses only the initial monomials. This could lead to confusing results for a non-homogeneous input.

If the optional second argument is non-zero, the Betti numbers will be minimized.
`betti` sets the attribute `rowShift`.

Example:

```
  ring r=32003,(a,b,c,d),dp;
  ideal j=bc-ad,b3-a2c,c3-bd2,ac2-b2d;
  list T=mres(j,0); // 0 forces a full resolution
  // a minimal set of generators for j:
  print(T[1]);
↦ bc-ad,
↦ c3-bd2,
↦ ac2-b2d,
↦ b3-a2c
  // second syzygy module of r/j which is the first
  // syzygy module of j (minimal generating set):
  print(T[2]);
↦ bd,c2,ac,b2,
↦ -a,-b,0, 0,
↦ c, d, -b,-a,
↦ 0, 0, -d,-c
  // the second syzygy module (minimal generating set):
  print(T[3]);
↦ -b,
↦ a,
↦ -c,
↦ d
  print(T[4]);
↦ 0
  betti(T);
↦ 1,0,0,0,
↦ 0,1,0,0,
↦ 0,3,4,1
  // most useful for reading off the graded Betti numbers:
  print(betti(T),"betti");
↦            0     1     2     3
↦ ------------------------------
↦     0:     1     -     -     -
↦     1:     -     1     -     -
↦     2:     -     3     4     1
↦ ------------------------------
↦ total:     1     4     4     1
↦
```

Hence,

- the 0th syzygy module of r/j (which is r) has 1 generator in degree 0 (which is 1),
- the 1st syzygy module `T[1]` (which is j) has 4 generators (one in degree 2 and three in degree 3),
- the 2nd syzygy module `T[2]` has 4 generators (all in degree 4),
- the 3rd syzygy module `T[3]` has 1 generator in degree 5,

where the generators are the columns of the displayed matrix and degrees are assigned such that the corresponding maps have degree 0:

$$0 \longleftarrow r/j \longleftarrow r(1) \xleftarrow{T[1]} r(2) \oplus r^3(3) \xleftarrow{T[2]} r^4(4) \xleftarrow{T[3]} r(5) \longleftarrow 0 \quad .$$

See Section C.3 [Syzygies and resolutions], page 703; Section 5.1.46 [fres], page 185; Section 5.1.56 [hres], page 195; Section 5.1.81 [lres], page 212; Section 5.1.96 [mres], page 223; Section 5.1.117 [print], page 238; [res], page 721; Section 4.18 [resolution], page 124; Section 5.1.145 [sres], page 264.

5.1.5 char

Syntax: `char` (ring_name)

Type: int

Purpose: returns the characteristic of the coefficient field of a ring.

Example:

```
  ring r=32003,(x,y),dp;
  char(r);
↦ 32003
  ring s=0,(x,y),dp;
  char(s);
↦ 0
  ring ra=(7,a),(x,y),dp;
  minpoly=a^3+a+1;
  char(ra);
↦ 7
  ring rp=(49,a),(x,y),dp;
  char(rp);
↦ 7
  ring rr=real,x,dp;
  char(rr);
↦ 0
```

See Section 5.1.7 [charstr], page 160; Section 4.19 [ring], page 125.

5.1.6 char_series

Syntax: `char_series` (ideal_expression)

Type: matrix

Purpose: the rows of the matrix represent the irreducible characteristic series of the ideal with respect to the current ordering of variables.
One application is the decomposition of the zero-set.

Example:

```
  ring r=32003,(x,y,z),dp;
  print(char_series(ideal(xyz,xz,y)));
↦ y,z,
↦ x,y
```

See Section C.4 [Characteristic sets], page 704.

5.1.7 charstr

Syntax: `charstr` (ring_name)

Type: string

Purpose: returns the description of the coefficient field of a ring. (Tests for certain types of coefficients should use the routines from `ring.lib` as the string representation may change.)

Example:

```
  ring r=32003,(x,y),dp;
  charstr(r);
↦ ZZ/32003
  ring s=0,(x,y),dp;
  charstr(s);
↦ QQ
  ring ra=(7,a),(x,y),dp;
  minpoly=a^3+a+1;
  charstr(ra);
↦ 7,a
  ring rp=(49,a),(x,y),dp;
  charstr(rp);
↦ 49,a
  ring rr=real,x,dp;
  charstr(rr);
↦ Float()
```

See Section 5.1.5 [char], page 159; Section 5.1.110 [ordstr], page 235; Section 4.19 [ring], page 125; Section D.2.11 [ring_lib], page 735; Section 5.1.162 [varstr], page 280.

5.1.8 chinrem

Syntax: `chinrem` (list, intvec)
`chinrem` (list, list)
`chinrem` (intvec, intvec)

Type: the same type as the elements of the first argument
If the elements of the first argument are lists again, chinrem is applied recusively.

Purpose: applies chinese remainder theorem to the first argument w.r.t. the moduli given in the second. The elements in the first list must be of same type which can be `bigint`/`int`, `poly`, `ideal`, `module` or `matrix`. The moduli, if given by a list, must be of type `bigint` or `int`.
If data depending on a ring are involved, the coeffcient field must be `Q`.

Example:

```
  chinrem(intvec(2,-3),intvec(7,11));
↦ 30
  chinrem(list(2,-3),list(7,11));
↦ 30
  ring r=0,(x,y),dp;
  ideal i1=5x+2y,x2+3y2+xy;
  ideal i2=2x-3y,2x2+4y2+5xy;
  chinrem(list(i1,i2),intvec(7,11));
```

```
↦ _[1]=-9x+30y
↦ _[2]=-20x2-6xy-18y2
  chinrem(list(i1,i2),list(bigint(7),bigint(11)));
↦ _[1]=-9x+30y
↦ _[2]=-20x2-6xy-18y2
  chinrem(list(list(i1,i2),list(i1,i2)),list(bigint(7),bigint(11)));
↦ [1]:
↦    _[1]=-9x+30y
↦    _[2]=-20x2-6xy-18y2
↦ [2]:
↦    _[1]=-9x+30y
↦    _[2]=-20x2-6xy-18y2
```

See Section D.4.13 [modstd_lib], page 748.

5.1.9 cleardenom

Syntax: `cleardenom` (poly_expression)
`cleardenom` (vector_expression)

Type: same as the input type

Purpose: multiplies a polynomial, resp. vector, by a suitable constant to cancel all denominators from its coefficients and then divide it by its content.

Example:

```
  ring r=0,(x,y,z),dp;
  poly f=(3x+6y)^5;
  f/5;
↦ 243/5x5+486x4y+1944x3y2+3888x2y3+3888xy4+7776/5y5
  cleardenom(f/5);
↦ x5+10x4y+40x3y2+80x2y3+80xy4+32y5
  vector w= [4x2+20,6x+2,0,8];   // application to a vector
  print(cleardenom(w));
↦ [2x2+10,3x+1,0,4]
```

See [content], page 733.

5.1.10 close

Syntax: `close` (link_expression)

Type: none

Purpose: closes a link.

Example:

```
link l="ssi:tcp localhost:"+system("Singular");
open(l); // start SINGULAR "server" on localhost in batchmode
close(l); // shut down SINGULAR server
```

See Section 4.9 [link], page 95; Section 5.1.107 [open], page 230.

5.1.11 coef

Syntax: `coef` (poly_expression, product_of_ringvars)

Type: matrix

Syntax: `coef` (vector_expression, product_of_ringvars, matrix_name, matrix_name)

Type: none

Purpose: determines the monomials in f divisible by a ring variable of m (where f is the first argument and m the second argument) and the coefficients of these monomials as polynomials in the remaining variables. First case: returns a $2 \times n$ matrix M, n being the number of the determined monomials. The first row consists of these monomials, the second row of the corresponding coefficients of the monomials in f. Thus, $f = M[1,1] \cdot M[2,1] + \ldots + M[1,n] \cdot M[2,n]$.

Second case: the second matrix (i.e., the 4th argument) contains the monomials, the first matrix (i.e., the 3rd argument) the corresponding coefficients of the monomials in the vector.

Note: coef considers only monomials which really occur in f (i.e., which are not 0), while coeffs (see Section 5.1.12 [coeffs], page 162) returns the coefficient 0 at the appropriate place if a monomial is not present.

Example:

```
  ring r=32003,(x,y,z),dp;
  poly f=x5+5x4y+10x2y3+y5;
  matrix m=coef(f,y);
  print(m);
↦ y5,y3,  y,  1,
↦ 1, 10x2,5x4,x5
  f=x20+xyz+xy+x2y+z3;
  print(coef(f,xy));
↦ x20,x2y,xy, 1,
↦ 1,  1,  z+1,z3
  vector v=[f,zy+77+xy];
  print(v);
↦ [x20+x2y+xyz+z3+xy,xy+yz+77]
  matrix mc; matrix mm;
  coef(v,y,mc,mm);
  print(mc);
↦ x2+xz+x,x20+z3,
↦ x+z,    77
  print(mm);
↦ y,1,
↦ y,1
```

See Section 5.1.12 [coeffs], page 162.

5.1.12 coeffs

Syntax: `coeffs` (poly_expression , ring_variable)
`coeffs` (ideal_expression, ring_variable)
`coeffs` (vector_expression, ring_variable)
`coeffs` (module_expression, ring_variable)

`coeffs` (poly_expression, ring_variable, matrix_name)
`coeffs` (ideal_expression, ring_variable, matrix_name)
`coeffs` (vector_expression, ring_variable, matrix_name)
`coeffs` (module_expression, ring_variable, matrix_name)

Type: matrix

Purpose: develops each polynomial of the first argument J as a univariate polynomial in the given ring_variable z, and returns the coefficients as a matrix M.

With e denoting the maximal z-degree occuring in the polynomials of J, and d:=e+1, M = (m_{ij}) satisfies the following conditions:

(i) If J is a single polynomial f, then M is a $(d \times 1)$-matrix and $m_{i+1,j}, 0 \leq i \leq e$, is the coefficient of z^i in f.

(ii) If J is an ideal with generators $f_1, f_2, \ldots, f_k$ then M is a $(d \times k)$-matrix and $m_{i+1,j}, 0 \leq i \leq e, 1 \leq j \leq k$, is the coefficient of z^i in f_j.

(iii) If J is a k-dimensional vector with entries $f_1, f_2, \ldots, f_k$ then M is a $(dk \times 1)$-matrix and $m_{(j-1)d+i+1,1}, 0 \leq i \leq e, 1 \leq j \leq k$, is the coefficient of z^i in f_j.

(iV) If J is a module generated by s vectors $v_1, v_2, \ldots, v_s$ of dimension k then M is a $(dk \times s)$-matrix and $m_{(j-1)d+i+1,r}, 0 \leq i \leq e, 1 \leq j \leq k, 1 \leq r \leq s$, is the coefficient of z^i in the j-th entry of v_r.

The optional third argument T can be used to return the matrix of powers of z such that matrix(J) = T*M holds in each of the previous four cases.

Note: `coeffs` returns the coefficient 0 at the appropriate matrix entry if a monomial is not present, while `coef` considers only monomials which actually occur in the given expression.

Example:

```
  ring r;
  poly f = (x+y)^3;
  poly g = xyz+z10y4;
  ideal i = f, g;
  matrix M = coeffs(i, y);
  print(M);
↦ x3, 0,
↦ 3x2,xz,
↦ 3x, 0,
↦ 1,  0,
↦ 0,  z10
  vector v = [f, g];
  M = coeffs(v, y);
  print(M);
↦ x3,
↦ 3x2,
↦ 3x,
↦ 1,
↦ 0,
↦ 0,
↦ xz,
↦ 0,
↦ 0,
↦ z10
```

Syntax: `coeffs` (ideal_expression, ideal_expression)
`coeffs` (module_expression, module_expression)
`coeffs` (ideal_expression, ideal_expression, product_of_ringvars)
`coeffs` (module_expression, module_expression, product_of_ringvars)

Type: matrix

Purpose: expresses each polynomial of the first argument M as a sum $\sum_{i=1}^{k} m_i \cdot a_i \cdot x^{e_i}$, where the m_i come from a specified set of monomials, the a_i are from the underlying coefficient ring (or field), and the x^{e_i} are powers of a specified ring variable x.

The second parameter K provides the set of monomials which should be sufficient to generate all entries of M.
Both M and K can be thought of as the matrices obtained by matrix(M) and matrix(K), respectively. (If M and K are given by ideals, then this matrix has just one row.)

The optional parameter product_of_ringvars determines the variable x: It is expected to be either the product of all ring variables (then x is 1, and each polynomial will be expressed as $\sum_{i=1}^{k} m_i \cdot a_i$, or product_of_ringvars is the product of all ring variables except one variable (which then determines x). If product_of_ringvars is omitted then x = 1 as default.

If K contains all monomials that are necessary to express the entries of M, then the returned matrix A satisfies $K \cdot A = M$. Otherwise only a subset of entries of $K \cdot A$ and M will coincide. In this case, the valid entries start at M[1,1] and run from left to right, top to bottom.

Note: Note that in general not all entries of K*A and M will coincide, depending on the set of monomials provided by K.

Example:

```
  ring r=32003,(x,y,z),dp;
  module M = [y3+x2z, xy], [-xy, y2+x2z];
  print(M);
↦ y3+x2z,-xy,
↦ xy,    x2z+y2
  module K = [x2, xy], [y3, xy], [xy, x];
  print(K);
↦ x2,y3,xy,
↦ xy,xy,x
  matrix A = coeffs(M, K, xy);   // leaving z as variable of interest
  print(A);   // attention: only the first row of M is reproduced by K*A
↦ z,0,
↦ 1,0,
↦ 0,-1
```

See Section 5.1.11 [coef], page 162; Section 5.1.67 [kbase], page 203.

5.1.13 contract

Syntax: `contract` (ideal_expression, ideal_expression)

Type: matrix

Purpose: contracts each of the n elements of the second ideal J by each of the m elements of the first ideal I, producing an $m \times n$ matrix.

Contraction is defined on monomials by:

$$\text{contract}(x^A, x^B) := \begin{cases} x^{(B-A)}, & \text{if } B \geq A \text{ componentwise} \\ 0, & \text{otherwise.} \end{cases}$$

where A and B are the multiexponents of the ring variables represented by x. `contract` is extended bilinearly to all polynomials.

Example:

```
ring r=0,(a,b,c,d),dp;
ideal I=a2,a2+bc,abc;
ideal J=a2-bc,abcd;
print(contract(I,J));
↦ 1,0,
↦ 0,ad,
↦ 0,d
```

See Section 5.1.23 [diff], page 169.

5.1.14 crossprod

Syntax: `crossprod (` cring_expression`, ... )`

Type: cring

Purpose: crooss product of several objects of type cring

Example:

```
crossprod(ZZ/32003,Float());
↦ ZZ/32003 x Float()
```

See Section 4.1 [cring], page 73.

5.1.15 datetime

Procedure from library `standard.lib` (see Section D.1 [standard_lib], page 721).

Syntax: `datetime ()`

Return: string

Purpose: return the curent date and time as a string

Example:

```
datetime();
↦ Di 13. Feb 13:19:18  2018
```

5.1.16 dbprint

Syntax: `dbprint (` int_expression`,` expression_list `)`

Type: none

Purpose: applies the print command to each expression in the expression_list if int_expression is positive. `dbprint` may also be used in procedures in order to print results subject to certain conditions.

Syntax: `dbprint (` expression `)`

Type: none

Purpose: The print command is applied to the expression if `printlevel>=voice`.

Note: See Section 3.8 [Libraries], page 55, for an example how this is used for displaying comments while procedures are executed.

Example:

```
  int debug=0;
  intvec i=1,2,3;
  dbprint(debug,i);
  debug=1;
  dbprint(debug,i);
↦ 1,
↦ 2,
↦ 3
  voice;
↦ 1
  printlevel;
↦ 0
  dbprint(i);
```

See Section 3.9 [Debugging tools], page 68; Section 5.1.117 [print], page 238; Section 5.3.6 [printlevel], page 299; Section 5.3.11 [voice], page 303.

5.1.17 defined

Syntax: `defined` (name)

Type: int

Purpose: returns a value `<>`0 (TRUE) if there is a user-defined object with this name, and 0 (FALSE) otherwise.
A non-zero return value is the level where the object is defined (level 1 denotes the top level, level 2 the level of a first procedure, level 3 the level of a procedure called by a first procedure, etc.). For ring variables and other constants, -1 is returned.

Note: A local object `m` may be identified by `if (defined(m)==voice)`.

Example:

```
  ring r=(0,t),(x,y),dp;
  matrix m[5][6]=x,y,1,2,0,x+y;
  defined(mm);
↦ 0
  defined(r) and defined(m);
↦ 1
  defined(m)==voice;   // m is defined in the current level
↦ 1
  defined(x);
↦ -1
  defined(z);
↦ 0
  defined("z");
↦ -1
  defined(t);
↦ -1
```

```
  defined(42);
↦ -1
```

See Section 5.1.135 [rvar], page 254; Section 5.3.11 [voice], page 303.

5.1.18 deg

Syntax: `deg` (poly_expression)
`deg` (vector_expression)
`deg` (poly_expression , intvec_expression)
`deg` (vector_expression , intvec_expression)

Type: int

Purpose: returns the maximal (weighted) degree of the terms of a polynomial or a vector; deg(0) is -1.
The optional second argument gives the weight vector, otherwise weight 1 is used for lex orderings and block ordering, the default weights of the base ring are used for orderings consisting of one block.

Example:

```
  ring r=0,(x,y,z),lp;
  deg(0);
↦ -1
  deg(x3+y4+xyz3);
↦ 5
  ring rr=7,(x,y),wp(2,3);
  poly f=x2+y3;
  deg(f);
↦ 9
  ring R=7,(x,y),ws(2,3);
  poly f=x2+y3;
  deg(f);
↦ 9
  vector v=[x2,y];
  deg(v);
↦ 4
```

See Section 5.1.66 [jet], page 202; Section 5.1.109 [ord], page 235; Section 4.16 [poly], page 118; Section 4.21 [vector], page 131.

5.1.19 degree

Syntax: `degree` (ideal_expression)
`degree` (module_expression)

Type: string

Purpose: computes the (Krull) dimension and the multiplicity of the ideal, resp. module, generated by the leading monomials of the input and prints it. This is equal to the dimension and multiplicity of the ideal, resp. module, if the input is a standard basis with respect to a degree ordering.

Example:

```
ring r3=32003,(x,y,z),ds;
int a,b,c,t=11,10,3,1;
poly f=x^a+y^b+z^(3*c)+x^(c+2)*y^(c-1)+x^(c-1)*y^(c-1)*z3
  +x^(c-2)*y^c*(y2+t*x)^2;
ideal i=jacob(f);
ideal i0=std(i);
degree(i0);
↦ // dimension (local)   = 0
↦ // multiplicity = 314
```

See Section 5.1.24 [dim], page 170; Section 4.5 [ideal], page 79; Section 5.1.98 [mult], page 224; Section 5.1.147 [std], page 266; Section 5.1.163 [vdim], page 280.

5.1.20 delete

Syntax: `delete` (list_expression, int_expression)
`delete` (intvec_expression, int_expression)
`delete` (ideal_expression, int_expression)
`delete` (module_expression, int_expression)

Type: type of the first argument

Purpose: deletes the element with the given index from a list/intvec/ideal/module (the input is not changed).

Example:

```
  list l="a","b","c";
  list l1=delete(l,2);l1;
↦ [1]:
↦    a
↦ [2]:
↦    c
  l;
↦ [1]:
↦    a
↦ [2]:
↦    b
↦ [3]:
↦    c
  delete(1..5,2);
↦ 1,3,4,5
  ring r=0,(x,y,z),dp;
  delete(maxideal(1),1);
↦ _[1]=y
↦ _[2]=z
```

See Section 4.5 [ideal], page 79; Section 5.1.60 [insert], page 197; Section 4.8 [intvec], page 92; Section 4.10 [list], page 102; Section 4.13 [module], page 111.

5.1.21 denominator

Syntax: `denominator` (number_expression)

Type: number

Purpose: returns the denominator of a number.

Example:

```
ring r = 0, x, dp;
number n = 3/2;
denominator(n);
↦ 2
```

See Section 5.1.9 [cleardenom], page 161; [content], page 733; Section 5.1.105 [numerator], page 229.

5.1.22 det

Syntax: `det` (intmat_expression)
`det` (matrix_expression)
`det` (module_expression)

Type: int, resp. poly

Purpose: returns the determinant of a square matrix. A module is considered as a matrix. The applied algorithms depend on type of input. If the input is a module or matrix with symbolic entries the Bareiss algorithm is used. In the other cases the chinese remainder algorithm is used. For large sparse problems the input as a module has advantages.

Example:

```
ring r=7,(x,y),wp(2,3);
matrix m[3][3]=1,2,3,4,5,6,7,8,x;
det(m);
↦ -3x-1
```

See Section 4.7 [intmat], page 89; Section 4.12 [matrix], page 107; Section 5.1.90 [minor], page 218.

5.1.23 diff

Syntax: `diff` (poly_expression, ring_variable)
`diff` (vector_expression, ring_variable)
`diff` (ideal_expression, ring_variable)
`diff` (module_expression, ring_variable)
`diff` (matrix_expression, ring_variable)

Type: the same as the type of the first argument

Syntax: `diff` (ideal_expression, ideal_expression)

Type: matrix

Syntax: `diff` (number_expression, ring_parameter)

Type: number

Purpose: computes the partial derivative of a polynomial object by a ring variable (first forms) respectively differentiates each polynomial (1..n) of the second ideal by the differential operator corresponding to each polynomial (1..m) in the first ideal, producing an m x n matrix.
respectively if the coefficient ring is a transcendental field extension, differentiates a number (that is, a rational function) by a transcendental variable (ring parameter).

Example:

```
  ring r=0,(x,y,z),dp;
  poly f=2x3y+3z5;
  diff(f,x);
↦ 6x2y
  vector v=[f,y2+z];
  diff(v,z);
↦ 15z4*gen(1)+gen(2)
  ideal j=x2-yz,xyz;
  ideal i=x2,x2+yz,xyz;
  // corresponds to differential operators
  // d2/dx2, d2/dx2+d2/dydz, d3/dxdydz:
  print(diff(i,j));
↦ 2,0,
↦ 1,x,
↦ 0,1
  // differentiation of rational functions:
  ring R=(0,t),(x),dp;
  number f = t^2/(1-t)^2;
  diff(f,t);
↦ (-2t)/(t3-3t2+3t-1)
```

See Section 5.1.13 [contract], page 164; Section 4.5 [ideal], page 79; Section 5.1.64 [jacob], page 200; Section 4.12 [matrix], page 107; Section 4.13 [module], page 111; Section 4.16 [poly], page 118; Section 5.1.160 [var], page 279; Section 4.21 [vector], page 131.

5.1.24 dim

Syntax: `dim` (ideal_expression)
`dim` (module_expression)
`dim` (resolution_expression)
`dim` (ideal_expression `,` ideal_expression)
`dim` (module_expression `,` ideal_expression)

Type: int

Purpose: computes the dimension of the ideal, resp. module, generated by the leading monomials of the given generators of the ideal, resp. module. This is also the dimension of the ideal if it is represented by a standard basis.
`dim(I,J)` is the dimension of `I/J`.
`dim(` res `)` computes the cohomological dimension of res[1].

Note: The dimension of an ideal I means the Krull dimension of the basering modulo I.
The dimension of a module is the dimension of its annihilator ideal.
In the case of ideal (1), -1 is returned.

Example:

```
  ring r=32003,(x,y,z),dp;
  ideal I=x2-y,x3;
  dim(std(I));
↦ 1
  dim(std(ideal(1)));
↦ -1
```

See Section 5.1.19 [degree], page 167; Section 4.5 [ideal], page 79; Section 5.1.98 [mult], page 224; Section 5.1.147 [std], page 266; Section 5.1.163 [vdim], page 280.

5.1.25 division

Syntax: `division` (ideal_expression, ideal_expression)
`division` (module_expression, module_expression)
`division` (ideal_expression, ideal_expression, int_expression)
`division` (module_expression, module_expression, int_expression)
`division` (ideal_expression, ideal_expression, int_expression, intvec_expression)
`division` (module_expression, module_expression, int_expression, intvec_expression)

Type: list

Purpose: `division` computes a division with remainder. For two ideals resp. modules `M` (first argument) and `N` (second argument), it returns a list `T,R,U` where `T` is a matrix, `R` is an ideal resp. a module, and `U` is a diagonal matrix of units such that `matrix(M)*U=matrix(N)*T+matrix(R)` is a standard representation for the normal form `R` of `M` with respect to a standard basis of `N`. `division` uses different algorithms depending on whether `N` is represented by a standard basis. For a polynomial basering, the matrix `U` is the identity matrix. A matrix `T` as above is also computed by `lift`.
For additional arguments `n` (third argument) and `w` (fourth argument), `division` returns a list `T,R` as above such that `matrix(M)=matrix(N)*T+matrix(R)` is a standard representation for the normal form `R` of `M` with respect to `N` up to weighted degree `n` with respect to the weight vector `w`. The weighted degree of `T` and `R` respect to `w` is at most `n`. If the weight vector `w` is not given, `division` uses the standard weight vector `w=1,...,1`.

Example:

```
ring R=0,(x,y),ds;
poly f=x5+x2y2+y5;
division(f,jacob(f)); // automatic conversion: poly -> ideal
↦ [1]:
↦    _[1,1]=1/5x
↦    _[2,1]=3/10y
↦ [2]:
↦    _[1]=-1/2y5
↦ [3]:
↦    _[1,1]=1
division(f^2,jacob(f));
↦ [1]:
↦    _[1,1]=1/20x6-9/80xy5-5/16x7y+5/8x2y6
↦    _[2,1]=1/8x2y3+1/5x5y+1/20y6-3/4x3y4-5/4x6y2-5/16xy7
↦ [2]:
↦    _[1]=0
↦ [3]:
↦    _[1,1]=1/4-25/16xy
division(ideal(f^2),jacob(f),10);
↦ // ** _ is no standard basis
↦ [1]:
↦    _[1,1]=-75/8y9
↦    _[2,1]=1/2x2y3+x5y-1/4y6-3/2x3y4+15/4xy7+375/16x2y8
↦ [2]:
↦    _[1]=x10+9/4y10
```

See Section 4.5 [ideal], page 79; Section 5.1.78 [lift], page 209; Section 4.13 [module], page 111; Section 4.16.3 [poly operations], page 120; Section 5.1.127 [reduce], page 246.

5.1.26 dump

Syntax: `dump` (link_expression)

Type: none

Purpose: dumps (i.e., writes in a "message" or "block") the state of the SINGULAR session (i.e., all defined variables and their values) to the specified link (which must be either an ASCII or ssi link) such that a `getdump` can retrieve it later on.

Example:

```
ring r;
// write the whole session to the file dump.ascii
// in ASCII format
dump(":w dump.ascii");
kill r;                  // kill the basering
// reread the session from the file
// redefining everything which was not explicitly killed before
getdump("dump.ascii");
↦ // ** redefining stdfglm (load("standard.lib","try");)
↦ // ** redefining stdfglm (load("standard.lib","try");)
↦ // ** redefining stdhilb (load("standard.lib","try");)
↦ // ** redefining stdhilb (load("standard.lib","try");)
↦ // ** redefining quotientList (load("standard.lib","try");)
↦ // ** redefining quotientList (load("standard.lib","try");)
↦ // ** redefining par2varRing (load("standard.lib","try");)
↦ // ** redefining par2varRing (load("standard.lib","try");)
↦ // ** redefining hilbRing (load("standard.lib","try");)
↦ // ** redefining hilbRing (load("standard.lib","try");)
↦ // ** redefining qslimgb (load("standard.lib","try");)
↦ // ** redefining qslimgb (load("standard.lib","try");)
↦ // ** redefining groebner (load("standard.lib","try");)
↦ // ** redefining groebner (load("standard.lib","try");)
↦ // ** redefining res (load("standard.lib","try");)
↦ // ** redefining res (load("standard.lib","try");)
↦ // ** redefining quot (load("standard.lib","try");)
↦ // ** redefining quot (load("standard.lib","try");)
↦ // ** redefining quot1 (load("standard.lib","try");)
↦ // ** redefining quotient0 (load("standard.lib","try");)
↦ // ** redefining quotient1 (load("standard.lib","try");)
↦ // ** redefining quotient1 (load("standard.lib","try");)
↦ // ** redefining quotient2 (load("standard.lib","try");)
↦ // ** redefining quotient2 (load("standard.lib","try");)
↦ // ** redefining quotient3 (load("standard.lib","try");)
↦ // ** redefining quotient3 (load("standard.lib","try");)
↦ // ** redefining quotient5 (load("standard.lib","try");)
↦ // ** redefining quotient5 (load("standard.lib","try");)
↦ // ** redefining quotient4 (load("standard.lib","try");)
↦ // ** redefining quotient4 (load("standard.lib","try");)
↦ // ** redefining intersect1 (load("standard.lib","try");)
↦ // ** redefining sprintf (load("standard.lib","try");)
```

```
↦ // ** redefining sprintf (load("standard.lib","try");)
↦ // ** redefining printf (load("standard.lib","try");)
↦ // ** redefining printf (load("standard.lib","try");)
↦ // ** redefining fprintf (load("standard.lib","try");)
↦ // ** redefining fprintf (load("standard.lib","try");)
↦ // ** redefining weightKB (load("standard.lib","try");)
↦ // ** redefining weightKB (load("standard.lib","try");)
↦ // ** redefining datetime (load("standard.lib","try");)
↦ // ** redefining datetime (load("standard.lib","try");)
↦ // ** redefining kbclass (load("standard.lib","try");)
↦ // ** redefining delkbclass (load("standard.lib","try");)
↦ // ** redefining kbprepare (load("standard.lib","try");)
↦ // ** redefining checkww (load("standard.lib","try");)
↦ // ** redefining widkbase (load("standard.lib","try");)
↦ // ** redefining weightmon (load("standard.lib","try");)
↦ // ** redefining max (load("standard.lib","try");)
↦ // ** redefining max (load("standard.lib","try");)
↦ // ** redefining min (load("standard.lib","try");)
↦ // ** redefining min (load("standard.lib","try");)
↦ // ** redefining mod_init (load("standard.lib","try");)
  r;
↦ // coefficients: ZZ/32003
↦ // number of vars : 3
↦ //        block   1 : ordering dp
↦ //                  : names    x y z
↦ //        block   2 : ordering C
```

Restrictions:
For ASCII links, integer matrices contained in lists are dumped as integer list elements (and not as integer matrices), and lists of lists are dumped as one flatted list. Furthermore, links themselves are not dumped.

See Section 5.1.50 [getdump], page 188; Section 4.9 [link], page 95; Section 5.1.169 [write], page 283.

5.1.27 eliminate

Syntax: `eliminate` (ideal_expression, product_of_ring_variables)
`eliminate` (module_expression, product_of_ring_variables)
`eliminate` (ideal_expression, intvec_expression)
`eliminate` (module_expression, intvec_expression)
`eliminate` (ideal_expression, product_of_ring_variables, intvec_hilb)
`eliminate` (module_expression, product_of_ring_variables, intvec_hilb)

Type: the same as the type of the first argument

Purpose: eliminates variables occurring as factors/entries of the second argument from an ideal (resp. a submodule of a free module), by intersecting it (resp. each component of the submodule) with the subring not containing these variables.
`eliminate` does not need a special ordering nor a standard basis as input.

Note: Since elimination is expensive, for homogeneous input it might be useful first to compute the Hilbert function of the ideal (first argument) with a fast ordering (e.g., `dp`). Then make use of it to speed up the computation: a Hilbert-driven elimination uses the intvec provided as the third argument.

If the ideal (resp. module) is not homogeneous with weights 1, this intvec will be silently ignored.

Example:

```
  ring r=32003,(x,y,z),dp;
  ideal i=x2,xy,y5;
  eliminate(i,x);
↦ _[1]=y5
  ring R=0,(x,y,t,s,z),dp;
  ideal i=x-t,y-t2,z-t3,s-x+y3;
  eliminate(i,ts);
↦ _[1]=y2-xz
↦ _[2]=xy-z
↦ _[3]=x2-y
  ideal j=x2,xy,y2;
  intvec v=hilb(std(j),1);
  eliminate(j,y,v);
↦ _[1]=x2
```

See Section 5.1.54 [hilb], page 193; Section 4.5 [ideal], page 79; Section 4.13 [module], page 111; Section 5.1.147 [std], page 266.

5.1.28 eval

Syntax: `eval` (expression)

Type: none

Purpose: evaluates (quoted) expressions. Within a quoted expression, the quote can be "undone" by an `eval` (i.e., each eval "undoes" the effect of exactly one quote). Used only when receiving a quoted expression from an ssi link, with `quote` and `write` to prevent local evaluations when writing to an ssi link.

Example:

```
  link l="ssi:w example.ssi";
  ring r=0,(x,y,z),ds;
  ideal i=maxideal(3);
  ideal j=x7+x3,x2,z;
  // compute i+j before writing, but not std
  // this writes 'std(ideal(x3,...,z))'
  write (l, quote(std(eval(i+j))));
  option(prot);
  close(l);
  // now read it in again and evaluate
  // read(l) forces to compute 'std(ideal(x3,...,z))'
  read(l);
↦ _[1]=z
↦ _[2]=x2
↦ _[3]=xy2
↦ _[4]=y3
  close(l);
```

See Section 4.9.5 [Ssi links], page 97; Section 5.1.122 [quote], page 243; Section 5.1.169 [write], page 283.

5.1.29 ERROR

Syntax: `ERROR` (string_expression)

Type: none

Purpose: Immediately interrupts the current computation, returns to the top-level, and displays the argument `string_expression` as error message.

Note: This should be used as an emergency, resp. failure, exit within procedures.

Example:

```
int i=1;
proc myError() {ERROR("Need to leave now");i=2;}
myError();
↦    ? Need to leave now
↦    ? leaving ::myError
i;
↦ 1
```

5.1.30 example

Syntax: `example` topic ;

Purpose: computes an example for `topic`. Examples are available for all SINGULAR kernel and library functions. Where available (e.g., within Emacs), use `<TAB>` completion for a list of all available example `topics`.

Example:

```
example prime;
example intvec_declarations;
```

Section 5.1.52 [help], page 191

5.1.31 execute

Syntax: `execute` (string_expression)

Type: none

Purpose: executes a string containing a sequence of SINGULAR commands.

Note: The command `return` cannot appear in the string.
`execute` should be avoided in procedures whenever possible, since it may give rise to name conflicts. Moreover, such procedures cannot be precompiled (a feature which SINGULAR will provide in the future).

Example:

```
  ring r=32003,(x,y,z),dp;
  ideal i=x+y,z3+22y;
  write(":w save_i",i);
  ring r0=0,(x,y,z),Dp;
  string s="ideal k="+read("save_i")+";";
  s;
↦ ideal k=x+y,z3+22y;
  execute(s); // define the ideal k
  k;
↦ k[1]=x+y
↦ k[2]=z3+22y
```

5.1.32 extgcd

Syntax: `extgcd` (int_expression, int_expression)
`extgcd` (bigint_expression, bigint_expression)
`extgcd` (poly_expression, poly_expression)

Type: list of 3 objects of the same type as the type of the arguments

Purpose: computes extended gcd: the first element is the greatest common divisor of the two arguments, the second and third are factors such that if `list L=extgcd(a,b);` then L[1]=a*L[2]+b*L[3].

Note: Polynomials must be univariate (in the same variable) to apply `extgcd`.

Example:

```
  extgcd(24,10);
↦ [1]:
↦    2
↦ [2]:
↦    -2
↦ [3]:
↦    5
  ring r=0,(x,y),lp;
  extgcd(x4-x6,(x2+x5)*(x2+x3));
↦ [1]:
↦    x5+x4
↦ [2]:
↦    1/2x2+1/2x+1/2
↦ [3]:
↦    1/2
```

See Section 5.1.48 [gcd], page 187; Section 4.6 [int], page 83.

5.1.33 facstd

Syntax: `facstd` (ideal_expression)
`facstd` (ideal_expression, ideal_expression)

Type: list of ideals

Purpose: returns a list of ideals computed by the factorizing Groebner basis algorithm.
The intersection of these ideals has the same zero-set as the input, i.e., the radical of the intersection coincides with the radical of the input ideal. In many (but not all!) cases this is already a decomposition of the radical of the ideal. (Note however that in general, no inclusion between the input and output ideals holds.)
The second, optional argument gives a list of polynomials which define non-zero constraints: those ideals which contain one of the constraint polynomials are omitted from the output list. Thus the zero set of the intersection of the output ideals is contained in the zero set V of the first input ideal and contains the complement in V of the zero set of the second input ideal.

Note: Not implemented for baserings over real ground fields and galois fields (that is, only implemented for ground fields for which Section 5.1.35 [factorize], page 178 is implemented).

Example:

```
  ring r=32003,(x,y,z),(c,dp);
  ideal I=xyz,x2z;
  facstd(I);
↦ [1]:
↦    _[1]=z
↦ [2]:
↦    _[1]=x
  facstd(I,x);
↦ [1]:
↦    _[1]=z
```

See Section 4.5 [ideal], page 79; Section 4.19 [ring], page 125; Section 5.1.147 [std], page 266.

5.1.34 factmodd

Syntax: `factmodd` (poly_expression, int_expression
[, poly_expression, poly_expression]
[, int_expression, int_expression]
)

Type: list of polys

Purpose: Computes a factorization of a polynomial h(x, y) in K[[x]][y] up to a certain degree in x, whenever a factorization of h(0, y) is provided or can be computed.
The algorithm is based on Hensel's lemma: Let h(x, y) denote a monic polynomial in y of degree m + n with coefficients in K[[x]]. Suppose there are two monic factors f_0(y) (of degree n) and g_0(y) of degree (m) such that
h(0, y) = f_0(y) * g_0(y) and <f_0, g_0> = K[y].
Fix an integer d >= 0. Then there are monic polynomials in y with coefficients in K[[x]], namely f(x, y) of degree n and g(x, y) of degree m such that
h(x, y) = f(x, y) * g(x, y) modulo <x^(d+1)> (*).
The function's six arguments are h, d, f_0, g_0, xIndex, and yIndex, where xIndex and yIndex denote indices of ring variables that are to play the roles of x and y as above.
h must be provided as an element of K[x,y] since all terms of h with x-degree larger than d can be ignored due to (*).
If f_0 and g_0 are not given, the algorithm computes the factorization of h(0, y) and is expected to find exactly two distinct factors (which may appear with multiplicities larger than 1) and uses these as f_0 and g_0.
If xIndex and yIndex are missing they will be expected to be 1 and 2, respectively.

Note: The function expects the ground ring to contain at least two variables.

Example:

```
  ring r = 0, (x,y), dp;
  poly f0 = y240; poly g0 = y102+1;
  poly h = y342+14x260+7x140y110+2x120y130+y240;
  int d = 260;
  list L = factmodd(h, d, f0, g0); L;
↦ [1]:
↦    -14x260y204-4x240y224-14x260y102-7x140y212-2x120y232+14x260+7x140y110+\
   2x120y130+y240
↦ [2]:
↦    42x260y66+8x240y86+7x140y74+2x120y94+y102+1
  // check result: next output should be zero
```

```
  reduce(h - L[1] * L[2], std(x^(d+1)));
↦ 0
```

See Section 5.1.35 [factorize], page 178.

5.1.35 factorize

Syntax: `factorize` (poly_expression)
`factorize` (poly_expression, `0`)
`factorize` (poly_expression, `2`)

Type: list of ideal and intvec

Syntax: `factorize` (poly_expression, `1`)

Type: ideal

Purpose: computes the irreducible factors (as an ideal) of the polynomial together with or without the multiplicities (as an intvec) depending on the second argument:
0: returns factors and multiplicities, first factor is a constant.
May also be written with only one argument.
1: returns non-constant factors (no multiplicities).
2: returns non-constant factors and multiplicities.

Note: Not implemented for the coefficient fields real, finite fields of type `(p^n,a)` and ZZ/m.

Example:

```
  ring r=32003,(x,y,z),dp;
  factorize(9*(x-1)^2*(y+z));
↦ [1]:
↦    _[1]=9
↦    _[2]=y+z
↦    _[3]=x-1
↦ [2]:
↦    1,1,2
  factorize(9*(x-1)^2*(y+z),1);
↦ _[1]=y+z
↦ _[2]=x-1
  factorize(9*(x-1)^2*(y+z),2);
↦ [1]:
↦    _[1]=y+z
↦    _[2]=x-1
↦ [2]:
↦    1,2
  ring rQ=0,x,dp;
  poly f = x2+1;            // irreducible in Q[x]
  factorize(f);
↦ [1]:
↦    _[1]=1
↦    _[2]=x2+1
↦ [2]:
↦    1,1
  ring rQi = (0,i),x,dp;
  minpoly = i2+1;
  poly f = x2+1;            // splits into linear factors in Q(i)[x]
```

```
  factorize(f);
↦ [1]:
↦    _[1]=1
↦    _[2]=x+(-i)
↦    _[3]=x+(i)
↦ [2]:
↦    1,1,1
```

See [absFactorize], page 741; Section 4.16 [poly], page 118.

5.1.36 farey

Syntax: `farey` (bigint_expression `,` bigint_expression `)`
`farey` (ideal_expression `,` bigint_expression `)`
`farey` (module_expression `,` bigint_expression `)`
`farey` (matrix_expression `,` bigint_expression `)`
`farey` (list_expression `,` bigint_expression `)`

Type: type of the first argument (unless it is `list`)

Purpose: lift the first argument modulo the second to the rationals.
The (coeffients of the) result a/b is the best approximation under the condition $|a|, |b| <= \sqrt{(N-1)/2}$ `farey(list(a,b,..),B)` is equivalent to `list(farey(a,B),farey(b,B),...)`.

Note: The current coefficient field must be the rationals.

Example:

```
  ring r=0,x,dp;
  farey(2,32003);
↦ 2
```

See Section 5.1.8 [chinrem], page 160.

5.1.37 fetch

Syntax: `fetch` (ring_name`,` name `)`
`fetch` (ring_name`,` name`,` intvec_expression `)`
`fetch` (ring_name`,` name`,` intvec_expression`,` intvec_expression `)`

Type: number, poly, vector, ideal, module, matrix or list (the same type as the second argument)

Purpose: maps objects between rings. `fetch` is the identity map between rings and qrings, in the first case the i-th variable of the source ring is mapped to the i-th variable of the basering. If the basering has less variables than the source ring these variables are mapped to zero. In the 2nd and 3rd form the intvec describes the permation of the variables: an i at position j maps the variable `var(j)` of the source to the variable `var(i)` of the destination. Negative numbers (and the fourth argument) describe mapping of parameters.
A zero means that that variable/parameter is mapped to 0.
The coefficient fields must be compatible. (See Section 4.11 [map], page 104 for a description of possible mappings between different ground fields).
`fetch` offers a convenient way to change variable names or orderings, or to map objects from a ring to a quotient ring of that ring or vice versa.

Note: Compared with `imap`, `fetch` uses the position of the ring variables, not their names.

Example:

```
  ring r=0,(x,y,z),dp;
  ideal i=maxideal(2);
  ideal j=std(i);
  poly f=x+y2+z3;
  vector v=[f,1];
  qring q=j;
  poly f=fetch(r,f);
  f;
↦ z3+y2+x
  vector v=fetch(r,v);
  v;
↦ z3*gen(1)+y2*gen(1)+x*gen(1)+gen(2)
  ideal i=fetch(r,i);
  i;
↦ i[1]=z2
↦ i[2]=yz
↦ i[3]=y2
↦ i[4]=xz
↦ i[5]=xy
↦ i[6]=x2
  ring rr=0,(a,b,c),lp;
  poly f=fetch(q,f);
  f;
↦ a+b2+c3
  vector v=fetch(r,v);
  v;
↦ a*gen(1)+b2*gen(1)+c3*gen(1)+gen(2)
  ideal k=fetch(q,i);
  k;
↦ k[1]=c2
↦ k[2]=bc
↦ k[3]=b2
↦ k[4]=ac
↦ k[5]=ab
↦ k[6]=a2
  fetch(q,i,1..nvars(q)); // equivalent to fetch(q,i)
↦ _[1]=c2
↦ _[2]=bc
↦ _[3]=b2
↦ _[4]=ac
↦ _[5]=ab
↦ _[6]=a2
```

See Section 5.1.57 [imap], page 196; Section 4.11 [map], page 104; Section 4.19.1 [qring], page 125; Section 4.19 [ring], page 125.

5.1.38 fglm

Syntax: `fglm` (ring_name, ideal_name)

Type: ideal

Purpose: computes for the given ideal in the given ring a reduced Groebner basis in the current ring, by applying the so-called FGLM (Faugere, Gianni, Lazard, Mora) algorithm.
The main application is to compute a lexicographical Groebner basis from a reduced Groebner basis with respect to a degree ordering. This can be much faster than computing a lexicographical Groebner basis directly.

Assume: The ideal must be zero-dimensional and given as a reduced Groebner basis in the given ring. The monomial ordering must be global.

Note: The only permissible differences between the given ring and the current ring are the monomial ordering and a permutation of the variables, resp. parameters.

Example:

```
  ring r=0,(x,y,z),dp;
  ideal i=y3+x2, x2y+x2, x3-x2, z4-x2-y;
  option(redSB);   // force the computation of a reduced SB
  i=std(i);
  vdim(i);
↦ 28
  ring s=0,(z,x,y),lp;
  ideal j=fglm(r,i);
  j;
↦ j[1]=y4+y3
↦ j[2]=xy3-y3
↦ j[3]=x2+y3
↦ j[4]=z4+y3-y
```

See Section 5.1.39 [fglmquot], page 181; Section 5.1.108 [option], page 230; Section 4.19.1 [qring], page 125; Section 4.19 [ring], page 125; Section 5.1.147 [std], page 266; [stdfglm], page 721; Section 5.1.163 [vdim], page 280.

5.1.39 fglmquot

Syntax: `fglmquot` (ideal_expression, poly_expression)

Type: ideal

Purpose: computes a reduced Groebner basis of the ideal quotient `I:p` of a zero-dimensional ideal `I` and a polynomial `p` using FGLM-techniques.

Assume: The ideal must be zero-dimensional and given as a reduced Groebner basis in the given ring. The polynomial must be reduced with respect to the ideal.

Example:

```
  ring r=0,(x,y,z),lp;
  ideal i=y3+x2,x2y+x2,x3-x2,z4-x2-y;
  option(redSB);   // force the computation of a reduced SB
  i=std(i);
  poly p=reduce(x+yz2+z10,i);
  ideal j=fglmquot(i,p);
  j;
↦ j[1]=z12
↦ j[2]=yz4-z8
↦ j[3]=y2+y-z8-z4
↦ j[4]=x+y-z10-z6-z4
```

See Section 5.1.38 [fglm], page 180; Section 5.1.108 [option], page 230; Section 5.1.123 [quotient], page 243; Section 4.19 [ring], page 125; Section 5.1.147 [std], page 266; Section 5.1.163 [vdim], page 280.

5.1.40 files, input from

Syntax: `<` "filename"

Type: none

Purpose: Read and execute the content of the file filename. Shorthand for `execute(read(filename))`.

Example:

```
< "example"; //read in the file example and execute it
```

See Section 5.1.31 [execute], page 175; Section 5.1.126 [read], page 245.

5.1.41 find

Syntax: `find (` string_expression`,` substring_expression `)`
`find (` string_expression`,` substring_expression`,` int_expression `)`

Type: int

Purpose: returns the first position of the substring in the string or 0 (if not found), starts the search at the position given in the 3rd argument.

Example:

```
  find("Aac","a");
↦ 2
  find("abab","a"+"b");
↦ 1
  find("abab","a"+"b",2);
↦ 3
  find("abab","ab",3);
↦ 3
  find("0123","abcd");
↦ 0
```

See Section 4.20 [string], page 128.

5.1.42 finduni

Syntax: `finduni (` ideal_expression `)`

Type: ideal

Purpose: returns an ideal which is contained in the ideal_expression, such that the i-th generator is a univariate polynomial in the i-th ring variable.
The polynomials have minimal degree w.r.t. this property.

Assume: The ideal must be zero-dimensional and given as a reduced Groebner basis in the current ring.

Example:

```
  ring  r=0,(x,y,z), dp;
  ideal i=y3+x2,x2y+x2,z4-x2-y;
  option(redSB);  // force computation of reduced basis
  i=std(i);
  ideal k=finduni(i);
  print(k);
↦ x4-x2,
↦ y4+y3,
↦ z12
```

See Section 5.1.108 [option], page 230; Section 4.19 [ring], page 125; Section 5.1.147 [std], page 266; Section 5.1.163 [vdim], page 280.

5.1.43 Float

Syntax: `Float ( )`
`Float (` int_expression `)`
`Float (` int_expression `,` int_expression `)`

Type: cring

Purpose: returns a coefficient ring of floating point (inexect) real number to be used in ring definitions.

Example:

```
  ring R1=Float(),(x,y),dp;
  R1;
↦ // coefficients: Float()
↦ // number of vars : 2
↦ //        block   1 : ordering dp
↦ //                  : names    x y
↦ //        block   2 : ordering C
  ring R2=Float(10,20),(a,b),dp;
  R2;
↦ // coefficients: Float(10,20)
↦ // number of vars : 2
↦ //        block   1 : ordering dp
↦ //                  : names    a b
↦ //        block   2 : ordering C
```

See Section 4.1 [cring], page 73; Section 4.19 [ring], page 125.

5.1.44 fprintf

Procedure from library `standard.lib` (see Section D.1 [standard_lib], page 721).

Syntax: `fprintf (` link_expression`,` string_expression `[,` any_expressions`] )`

Return: none

Purpose: `fprintf(l,fmt,...);` performs output formatting. The second argument is a format control string. Additional arguments may be required, depending on the content of the control string. A series of output characters is generated as directed by the control string; these characters are written to the link l. The control string `fmt` is simply text to be copied, except that the string may contain conversion specifications.
Type `help print;` for a listing of valid conversion specifications. As an addition to

the conversions of `print`, the `%n` and `%2` conversion specification does not consume an additional argument, but simply generates a newline character.

Note: If one of the additional arguments is a list, then it should be enclosed once more into a `list()` command, since passing a list as an argument flattens the list by one level.

Example:

```
  ring r=0,(x,y,z),dp;
module m=[1,y],[0,x+z];
intmat M=betti(mres(m,0));
list l=r,m,M;
link li="";   // link to stdout
fprintf(li,"s:%s,l:%l",1,2);
↦ s:1,l:int(2)
fprintf(li,"s:%s",l);
↦ s:(QQ),(x,y,z),(dp(3),C)
fprintf(li,"s:%s",list(l));
↦ s:(QQ),(x,y,z),(dp(3),C),y*gen(2)+gen(1),x*gen(2)+z*gen(2),1,1
fprintf(li,"2l:%2l",list(l));
↦ 2l:list("(QQ),(x,y,z),(dp(3),C)",
↦ module(y*gen(2)+gen(1),
↦ x*gen(2)+z*gen(2)),
↦ intmat(intvec(1,1 ),1,2))
↦
fprintf(li,"%p",list(l));
↦ [1]:
↦    // coefficients: QQ
↦ // number of vars : 3
↦ //        block   1 : ordering dp
↦ //                  : names    x y z
↦ //        block   2 : ordering C
↦ [2]:
↦    _[1]=y*gen(2)+gen(1)
↦    _[2]=x*gen(2)+z*gen(2)
↦ [3]:
↦    1,1
fprintf(li,"%;",list(l));
↦ [1]:
↦    // coefficients: QQ
↦ // number of vars : 3
↦ //        block   1 : ordering dp
↦ //                  : names    x y z
↦ //        block   2 : ordering C
↦ [2]:
↦    _[1]=y*gen(2)+gen(1)
↦    _[2]=x*gen(2)+z*gen(2)
↦ [3]:
↦    1,1
↦
fprintf(li,"%b",M);
↦            0     1
↦ ------------------
↦     0:     1     1
↦ ------------------
```

```
↦ total:     1     1
↦
```

See also: Section 5.1.117 [print], page 238; [printf], page 721; [sprintf], page 721; Section 4.20 [string], page 128.

5.1.45 freemodule

Syntax: `freemodule` (int_expression)

Type: module

Purpose: creates the free module of rank n generated by `gen(1)`, . . . , `gen(n)`.

Example:

```
  ring r=32003,(x,y),(c,dp);
  freemodule(3);
↦ _[1]=[1]
↦ _[2]=[0,1]
↦ _[3]=[0,0,1]
  matrix m=freemodule(3); // generates the 3x3 unit matrix
  print(m);
↦ 1,0,0,
↦ 0,1,0,
↦ 0,0,1
```

See Section 5.1.49 [gen], page 187; Section 4.13 [module], page 111.

5.1.46 fres

Syntax: `fres` (ideal_expression/module_expression , int_expression , [string_expression])

Type: resolution

Purpose: computes a (not necessarily minimal) free resolution of the input ideal/module, using Schreyer's algorithm, see reference.
If the second argument is `n > 0`, then the resolution is computed up to step `n`. If it is `0`, `fres` computes the whole resolution.
The optional third argument can be set to

- `"complete"` (default) to compute the whole syzygy module in each step,
- `"frame"` to compute only the so-called frame,
- `"extended frame"` to compute only the first two terms of each generator w.r.t. the induced monomial ordering, or
- `"single module"` to return only the frame of each module except the last one and to return the last module in its entirety. This option can be used to reduce the amount of memory needed for the computation.

Note: The input ideal/module must be a standard basis.

Reference:

B. Erocal, O. Motsak, F.-O. Schreyer, A. Steenpass: Refined Algorithms to Compute Syzygies. J. Symb. Comput. 74 (2016), 308-327. `http://arxiv.org/abs/1502.01654`

Example:

```
  ring r = 0, (w,x,y,z), dp;
  ideal I = w2-xz, wx-yz, x2-wy, xy-z2, y2-wz;
  attrib(I, "isSB", 1);
  resolution s = fres(I, 0);
  s;
↦  1      5      6      2
↦ r <--  r <--  r <--  r
↦
↦ 0      1      2      3
↦ resolution not minimized yet
↦
  print(betti(s, 0), "betti");
↦            0     1     2     3
↦ ------------------------------
↦     0:     1     -     -     -
↦     1:     -     5     5     1
↦     2:     -     -     1     1
↦ ------------------------------
↦ total:     1     5     6     2
↦
  list l = s;
  print(l[1]);
↦ w2-xz,
↦ wx-yz,
↦ x2-wy,
↦ xy-z2,
↦ y2-wz
  print(l[2]);
↦ -x,y, 0, -z,0, -y2+wz,
↦ w, -x,-y,0, z, z2,
↦ -z,w, 0, -y,0, 0,
↦ 0, 0, w, x, -y,-yz,
↦ 0, 0, -z,-w,x, w2
  print(l[3]);
↦ 0, -y2+wz,
↦ y, z2,
↦ -x,-wy,
↦ w, yz,
↦ -z,-w2,
↦ 1, x
```

See Section A.3.4 [Free resolution], page 648; Section 5.1.91 [minres], page 220; [res], page 721; Section 5.1.145 [sres], page 264; Section 5.1.152 [syz], page 275.

5.1.47 frwalk

Syntax: `frwalk` (ring_name, ideal_name)
`frwalk` (ring_name, ideal_name , int_expression)

Type: ideal

Purpose: computes for the ideal `ideal_name` in the ring `ring_name` a Groebner basis in the current ring, by applying the fractal walk algorithm.
The main application is to compute a lexicographical Groebner basis from a reduced

Groebner basis with respect to a degree ordering. This can be much faster than computing a lexicographical Groebner basis directly.

Note: When calling `frwalk`, the only permissible difference between the ring `ring_name` and the active base ring is the monomial ordering.

Example:

```
  ring r=0,(x,y,z),dp;
  ideal i=y3+x2, x2y+x2, x3-x2, z4-x2-y;
  i=std(i);
  ring s=0,(x,y,z),lp;
  ideal j=frwalk(r,i);
  j;
↦ j[1]=z12
↦ j[2]=yz4-z8
↦ j[3]=y2+y-z8-z4
↦ j[4]=xy-xz4-y+z4
↦ j[5]=x2+y-z4
```

See Section 5.1.38 [fglm], page 180; [groebner], page 721; Section 4.19.1 [qring], page 125; Section 4.19 [ring], page 125; Section 5.1.147 [std], page 266.

5.1.48 gcd

Syntax: `gcd (` int_expression`,` int_expression `)`
`gcd (` bigint_expression`,` bigint_expression `)`
`gcd (` number_expression`,` number_expression `)`
`gcd (` poly_expression`,` poly_expression `)`

Type: the same as the type of the arguments

Purpose: computes the greatest common divisor.

Note: Not implemented for the coefficient fields real and finite fields of type `(p^n,a)`.
The gcd of two numbers is their gcd as integer numbers or polynomials, otherwise it is not defined.

Example:

```
  gcd(2,3);
↦ 1
  gcd(bigint(2)^20,bigint(3)^23);     // also applicable for bigints
↦ 1
  typeof(_);
↦ bigint
  ring r=0,(x,y,z),lp;
  gcd(3x2*(x+y),9x*(y2-x2));
↦ x2+xy
  gcd(number(6472674604870),number(878646537247372));
↦ 2
```

See Section 4.2 [bigint], page 74; Section 5.1.32 [extgcd], page 176; Section 4.6 [int], page 83; Section 4.14 [number], page 114.

5.1.49 gen

Syntax: `gen (` int_expression `)`

Type: vector

Purpose: returns the i-th free generator of a free module.

Example:

```
  ring r=32003,(x,y,z),(c,dp);
  gen(3);
↦ [0,0,1]
  vector v=gen(5);
  poly f=xyz;
  v=v+f*gen(4); v;
↦ [0,0,0,xyz,1]
  ring rr=32003,(x,y,z),dp;
  fetch(r,v);
↦ xyz*gen(4)+gen(5)
```

See Section 5.1.45 [freemodule], page 185; Section 4.6 [int], page 83; Section 4.21 [vector], page 131.

5.1.50 getdump

Syntax: `getdump` (link_expression)

Type: none

Purpose: reads the content of the entire file, resp. link, and restores all variables from it. For ASCII links, `getdump` is equivalent to an `execute(read(` link `))` command. For ssi links, `getdump` should only be used on data which were previously `dump`'ed.

Example:

```
  int i=3;
  dump(":w example.txt");
  kill i;
  option(noredefine);
  getdump("example.txt");
↦ // ** redefining stdfglm (load("standard.lib","try");)
↦ // ** redefining stdfglm (load("standard.lib","try");)
↦ // ** redefining stdhilb (load("standard.lib","try");)
↦ // ** redefining stdhilb (load("standard.lib","try");)
↦ // ** redefining quotientList (load("standard.lib","try");)
↦ // ** redefining quotientList (load("standard.lib","try");)
↦ // ** redefining par2varRing (load("standard.lib","try");)
↦ // ** redefining par2varRing (load("standard.lib","try");)
↦ // ** redefining hilbRing (load("standard.lib","try");)
↦ // ** redefining hilbRing (load("standard.lib","try");)
↦ // ** redefining qslimgb (load("standard.lib","try");)
↦ // ** redefining qslimgb (load("standard.lib","try");)
↦ // ** redefining groebner (load("standard.lib","try");)
↦ // ** redefining groebner (load("standard.lib","try");)
↦ // ** redefining res (load("standard.lib","try");)
↦ // ** redefining res (load("standard.lib","try");)
↦ // ** redefining quot (load("standard.lib","try");)
↦ // ** redefining quot (load("standard.lib","try");)
↦ // ** redefining quot1 (load("standard.lib","try");)
↦ // ** redefining quotient0 (load("standard.lib","try");)
↦ // ** redefining quotient1 (load("standard.lib","try");)
```

```
↦ // ** redefining quotient1 (load("standard.lib","try");)
↦ // ** redefining quotient2 (load("standard.lib","try");)
↦ // ** redefining quotient2 (load("standard.lib","try");)
↦ // ** redefining quotient3 (load("standard.lib","try");)
↦ // ** redefining quotient3 (load("standard.lib","try");)
↦ // ** redefining quotient5 (load("standard.lib","try");)
↦ // ** redefining quotient5 (load("standard.lib","try");)
↦ // ** redefining quotient4 (load("standard.lib","try");)
↦ // ** redefining quotient4 (load("standard.lib","try");)
↦ // ** redefining intersect1 (load("standard.lib","try");)
↦ // ** redefining sprintf (load("standard.lib","try");)
↦ // ** redefining sprintf (load("standard.lib","try");)
↦ // ** redefining printf (load("standard.lib","try");)
↦ // ** redefining printf (load("standard.lib","try");)
↦ // ** redefining fprintf (load("standard.lib","try");)
↦ // ** redefining fprintf (load("standard.lib","try");)
↦ // ** redefining weightKB (load("standard.lib","try");)
↦ // ** redefining weightKB (load("standard.lib","try");)
↦ // ** redefining datetime (load("standard.lib","try");)
↦ // ** redefining datetime (load("standard.lib","try");)
↦ // ** redefining kbclass (load("standard.lib","try");)
↦ // ** redefining delkbclass (load("standard.lib","try");)
↦ // ** redefining kbprepare (load("standard.lib","try");)
↦ // ** redefining checkww (load("standard.lib","try");)
↦ // ** redefining widkbase (load("standard.lib","try");)
↦ // ** redefining weightmon (load("standard.lib","try");)
↦ // ** redefining max (load("standard.lib","try");)
↦ // ** redefining max (load("standard.lib","try");)
↦ // ** redefining min (load("standard.lib","try");)
↦ // ** redefining min (load("standard.lib","try");)
↦ // ** redefining mod_init (load("standard.lib","try");)
  i;
↦ 3
```

Restrictions:
`getdump` is not supported for DBM links, or for a link connecting to `stdin` (standard input).

See Section 5.1.26 [dump], page 172; Section 4.9 [link], page 95; Section 5.1.126 [read], page 245.

5.1.51 groebner

Procedure from library `standard.lib` (see Section D.1 [standard_lib], page 721).

Syntax: `groebner` (ideal_expression)
`groebner` (module_expression)
`groebner` (ideal_expression, list of string_expressions)
`groebner` (ideal_expression, list of string_expressions and int_expression)

Type: type of the first argument

Purpose: computes a standard basis of the first argument `I` (ideal or module) by a heuristically chosen method (default) or by a method specified by further arguments of type string. Possible methods are:
- the direct methods `"std"` or `"slimgb"` without conversion,

- conversion methods "hilb" or "fglm" where a Groebner basis is first computed with an "easy" ordering and then converted to the ordering of the basering by the Hilbert driven Groebner basis computation or by linear algebra. The actual computation of the Groebner basis can be specified by "std" or by "slimgb" (not for all orderings implemented).
A further string "par2var" converts parameters to an extra block of variables before a Groebner basis computation (and afterwards back). option(prot) informs about the chosen method.

Hint: Since there exists no uniform best method for computing standard bases, and since the difference in performance of a method on different examples can be huge, it is recommended to test, for hard examples, first various methods on a simplified example (e.g. use characteristic 32003 instead of 0 or substitute a subset of parameters/variables by integers, etc.).

Example:

```
  intvec opt = option(get);
option(prot);
ring r  = 0,(a,b,c,d),dp;
ideal i = a+b+c+d,ab+ad+bc+cd,abc+abd+acd+bcd,abcd-1;
groebner(i);
↦ std in (QQ),(a,b,c,d),(dp(4),C)
↦ [65535:2]1(3)s2(2)s3s4-s5ss6-s7--
↦ product criterion:8 chain criterion:5
↦ _[1]=a+b+c+d
↦ _[2]=b2+2bd+d2
↦ _[3]=bc2+c2d-bd2-d3
↦ _[4]=bcd2+c2d2-bd3+cd3-d4-1
↦ _[5]=bd4+d5-b-d
↦ _[6]=c3d2+c2d3-c-d
↦ _[7]=c2d4+bc-bd+cd-2d2
ring s  = 0,(a,b,c,d),lp;
ideal i = imap(r,i);
groebner(i,"hilb");
↦ compute hilbert series with std in ring (QQ),(a,b,c,d,@),(dp(5),C,L(10485\
   75))
↦ weights used for hilbert series: 1,1,1,1,1
↦ [4095:1]1(3)s2(2)s3s4-s5ss6-s7--
↦ product criterion:8 chain criterion:5
↦ std with hilb in (QQ),(a,b,c,d,@),(lp(4),dp(1),C,L(1048575))
↦ [4095:1]1(6)s2(5)s3(4)s4-s5sshh6(3)shhhhh8shh
↦ product criterion:9 chain criterion:8
↦ hilbert series criterion:9
↦ dehomogenization
↦ simplification
↦ imap to ring (QQ),(a,b,c,d),(lp(4),C)
↦ _[1]=c2d6-c2d2-d4+1
↦ _[2]=c3d2+c2d3-c-d
↦ _[3]=bd4-b+d5-d
↦ _[4]=bc-bd5+c2d4+cd-d6-d2
↦ _[5]=b2+2bd+d2
↦ _[6]=a+b+c+d
```

```
ring R  = (0,a),(b,c,d),lp;
minpoly = a2+1;
ideal i = a+b+c+d,ab+ad+bc+cd,abc+abd+acd+bcd,d2-c2b2;
groebner(i,"par2var","slimgb");
↦ //add minpoly to input
↦ compute hilbert series with slimgb in ring (QQ),(b,c,d,a,@),(dp(5),C,L(10\
   48575))
↦ weights used for hilbert series: 1,1,1,1,1
↦ slimgb in ring (QQ),(b,c,d,a,@),(dp(5),C,L(1048575))
↦ CC2M[2,2](2)C3M[1,1](2)4M[2,e1](2)C5M[2,e2](3)C6M[1,1](0)
↦ NF:8 product criterion:15, ext_product criterion:3
↦ std with hilb in (QQ),(b,c,d,a,@),(lp(3),dp(1),dp(1),C,L(1048575))
↦ [4095:1]1(7)s2(6)s(5)s3(4)s4-s5sshh6(3)shhhhh
↦ product criterion:15 chain criterion:5
↦ hilbert series criterion:7
↦ dehomogenization
↦ simplification
↦ imap to ring (QQ),(b,c,d,a),(lp(3),dp(1),C)
↦ //simplification
↦ (S:4)rtrtrtr
↦ //imap to original ring
↦ _[1]=d2
↦ _[2]=c+(a)
↦ _[3]=b+c+d+(a)
groebner(i,"fglm");          //computes a reduced standard basis
↦ std in (0,a),(b,c,d),(dp(3),C)
↦ [65535:2]1(3)s2(2)s3s4-s5ss6-s7
↦ (S:2)--
↦ product criterion:9 chain criterion:1
↦ ..+++--
↦ vdim= 2
↦ ..++-+-
↦ _[1]=d2
↦ _[2]=c+(a)
↦ _[3]=b+d
option(set,opt);
```

See also: Section D.15.12 [ffmodstd_lib], page 837; Section D.4.13 [modstd_lib], page 748; Section D.15.28 [nfmodstd_lib], page 858; Section 5.1.141 [slimgb], page 261; Section 5.1.147 [std], page 266; [stdfglm], page 721; [stdhilb], page 721.

5.1.52 help

Syntax: `help;`
`help` topic ;

Type: none

Purpose: displays online help information for `topic` using the currently set help browser. If no `topic` is given, the title page of the manual is displayed.

Note:

- `?` may be used instead of `help`.

- `topic` can be an index entry of the SINGULAR manual or the name of a (loaded) procedure which has a help section.
- `topic` may contain wildcard characters (i.e., `*` characters).
- If a (possibly "wildcarded") `topic` cannot be found (or uniquely matched) a warning is displayed and no help information is provided.
- If `topic` is the name of a (loaded) procedure whose help section has changed w.r.t. the help available in the manual then, instead of displaying the respective help section of the manual in the help browser, the "newer" help section of the procedure is simply printed to the terminal.
- The browser in which the help information is displayed can be either set with the command-line option `--browser=<browser>` (see Section 3.1.6 [Command line options], page 19), or with the command `system("--browser", "<browser>")`. Use the command `system("browsers");` for a list of all available browsers. See Section 3.1.3 [The online help system], page 15, for more details about help browsers.

Example:

```
help;      // display title page of manual
help ring; // display help for 'ring'
?ringe;    // equivalent to 'help ringe;'
↦ // ** No help for topic 'ringe' (not even for '*ringe*')
↦ // ** Try '?;'       for general help
↦ // ** or  '?Index;'  for all available help topics
?ring*;
↦ //  ** No unique help for 'ring*'
↦ //  ** Try one of
↦ ?Rings and orderings; ?Rings and standard bases; ?ring;
↦ ?ring declarations; ?ring operations; ?ring related functions;
↦ ?ring.lib; ?ring_lib; ?ringtensor; ?ringweights;
help Rings and orderings;
help standard.lib;  // displays help for library 'standard.lib'
```

See Section 3.1.6 [Command line options], page 19; Section 3.8 [Libraries], page 55; Section 3.7.1 [Procedure definition], page 51; Section 3.1.3 [The online help system], page 15; Section 5.1.151 [system], page 270.

5.1.53 highcorner

Syntax: `highcorner` (ideal_expression)
`highcorner` (module_expression)

Type: poly, resp. vector

Purpose: returns the smallest monomial not contained in the ideal, resp. module, generated by the initial terms of the given generators. If the generators are a standard basis, this is also the smallest monomial not contained in the ideal, resp. module.
If the ideal, resp. module, is not zero-dimensional, 0 is returned.
The command works also in global orderings, but is not very useful there.

Note: Let the ideal I be given by a standard basis. Then `highcorner(I)` returns 0 if and only if `dim(I)>0` or `dim(I)=-1`. Otherwise it returns the smallest monomial m not in I which has the following properties (with x_i the variables of the basering):

- if $x_i > 1$ then x_i does not divide m (hence, m=1 if the ordering is global)

- given any set of generators $f_1, \ldots, f_k$ of I, let f_i' be obtained from f_i by deleting the terms divisible by $x_i \cdot m$ for all i with $x_i < 1$. Then $f_1', \ldots, f_k'$ generate I.

Example:

```
ring r=0,(x,y),ds;
ideal i=x3,x2y,y3;
highcorner(std(i));
↦ xy2
highcorner(std(ideal(1)));
↦ 0
```

See Section 5.1.24 [dim], page 170; Section 5.1.147 [std], page 266; Section 5.1.163 [vdim], page 280.

5.1.54 hilb

Syntax: `hilb` (ideal_expression)
`hilb` (module_expression)
`hilb` (ideal_expression`,` int_expression)
`hilb` (module_expression`,` int_expression)
`hilb` (ideal_expression`,` int_expression `,` intvec_expression)
`hilb` (module_expression`,` int_expression `,` intvec_expression)

Type: none (if called with one argument)
intvec (if called with two or three arguments)

Purpose: computes the (weighted) Hilbert series of the base ring R modulo the ideal, resp. R^k modulo the module, defined by the leading terms of the generators of the given ideal, resp. module.
If `hilb` is called with one argument, then the first and second Hilbert series together with some additional information are displayed.
If `hilb` is called with two arguments, then the n-th Hilbert series is returned as an intvec, where n = 1, 2 is the second argument.
If a weight vector w is a given as 3rd argument, then the Hilbert series is computed w.r.t. these weights w (by default all weights are set to 1).

Caution: The last entry of the returned intvec is not part of the actual Hilbert series, but is used in the Hilbert driven standard basis computation (see [stdhilb], page 721). (It is the minimum weight of the module generators or 0).

Syntax: `hilb` (intvec_expression)

Type: intvec

Purpose: computes the second Hilbert series from the first, i.e. if `intvec v=hilb(I,1);` then `hilb(v)` yields the same result as `hilb(I,2)`.

Note: If the input is homogeneous w.r.t. the weights and a standard basis, the result is the (weighted) Hilbert series of the original ideal, resp. module.

Example:

```
  ring R=32003,(x,y,z),dp;
  ideal i=x2,y2,z2;
  ideal s=std(i);
  hilb(s);
↦ //         1 t^0
```

```
↦ //        -3 t^2
↦ //         3 t^4
↦ //        -1 t^6
↦
↦ //         1 t^0
↦ //         3 t^1
↦ //         3 t^2
↦ //         1 t^3
↦ // dimension (affine) = 0
↦ // degree (affine)  = 8
  hilb(s,1);
↦ 1,0,-3,0,3,0,-1,0
  hilb(s,2);
↦ 1,3,3,1,0
  intvec w=2,2,2;
  hilb(s,1,w);
↦ 1,0,0,0,-3,0,0,0,3,0,0,0,-1,0
```

See Section C.2 [Hilbert function], page 702; Section 4.5 [ideal], page 79; Section 4.8 [intvec], page 92; Section 4.13 [module], page 111; Section 5.1.147 [std], page 266; [stdhilb], page 721.

5.1.55 homog

Syntax: `homog` (ideal_expression)
`homog` (module_expression)

Type: int

Purpose: tests for homogeneity: returns 1 for homogeneous input, 0 otherwise.

Note: If the current ring has a weighted monomial ordering, `homog` tests for weighted homogeneity w.r.t. the given weights.

Syntax:
`homog` (polynomial_expression, ring_variable)
`homog` (vector_expression, ring_variable)
`homog` (ideal_expression, ring_variable)
`homog` (module_expression, ring_variable)

Type: same as first argument

Purpose: homogenizes polynomials, vectors, ideals, or modules by multiplying each monomial with a suitable power of the given ring variable.

Note: If the current ring has a weighted monomial ordering, `homog` computes the weighted homogenization w.r.t. the given weights.
The homogenizing variable must have weight 1.

Example:

```
  ring r=32003,(x,y,z),ds;
  poly s1=x3y2+x5y+3y9;
  poly s2=x2y2z2+3z8;
  poly s3=5x4y2+4xy5+2x2y2z3+y7+11x10;
  ideal i=s1,s2,s3;
  homog(s2,z);
↦ x2y2z4+3z8
```

```
  homog(i,z);
↦ _[1]=3y9+x5yz3+x3y2z4
↦ _[2]=x2y2z4+3z8
↦ _[3]=11x10+y7z3+5x4y2z4+4xy5z4+2x2y2z6
  homog(i);
↦ 0
  homog(homog(i,z));
↦ 1
```

See Section 4.5 [ideal], page 79; Section 4.13 [module], page 111; Section 4.16 [poly], page 118; Section 4.21 [vector], page 131.

5.1.56 hres

Syntax: `hres` (ideal_expression`,` int_expression)

Type: resolution

Purpose: computes a free resolution of an ideal using the Hilbert-driven algorithm.

More precisely, let R be the basering and I be the given ideal. Then `hres` computes a minimal free resolution of R/I

$$... \longrightarrow F_2 \xrightarrow{A_2} F_1 \xrightarrow{A_1} R \longrightarrow R/I \longrightarrow 0.$$

If the int_expression k is not zero then the computation stops after k steps and returns a list of modules $M_i =$ `module`(A_i), i=1..k.

`list L=hres(I,0);` returns a list L of n modules (where n is the number of variables of the basering) such that `L[i]` $= M_i$ in the above notation.

Note: The ideal_expression has to be homogeneous.
Accessing single elements of a resolution may require some partial computations to be finished. Therefore, it may take some time.

Example:

```
  ring r=0,(x,y,z),dp;
  ideal I=xz,yz,x3-y3;
  def L=hres(I,0);
  print(betti(L),"betti");
↦            0     1     2
↦ ------------------------
↦     0:     1     -     -
↦     1:     -     2     1
↦     2:     -     1     1
↦ ------------------------
↦ total:     1     3     2
↦
  L[2];     // the first syzygy module of r/I
↦ _[1]=-x*gen(1)+y*gen(2)
↦ _[2]=-x2*gen(2)+y2*gen(1)+z*gen(3)
```

See Section 5.1.4 [betti], page 157; Section 5.1.46 [fres], page 185; Section 4.5 [ideal], page 79; Section 4.6 [int], page 83; Section 5.1.81 [lres], page 212; Section 5.1.91 [minres], page 220; Section 4.13 [module], page 111; Section 5.1.96 [mres], page 223; [res], page 721; Section 5.1.145 [sres], page 264.

5.1.57 imap

Syntax: `imap` (ring_name, name)

Type: number, poly, vector, ideal, module, matrix or list (the same type as the second argument)

Purpose: identity map on common subrings. `imap` is the map between rings and qrings with compatible ground fields which is the identity on variables and parameters of the same name and 0 otherwise. (See Section 4.11 [map], page 104 for a description of possible mappings between different ground fields). Useful for mapping from a homogenized ring to the original ring or for mappings from/to rings with/without parameters. Compared with `fetch`, `imap` uses the names of variables and parameters. Unlike `map` and `fetch` `imap` can map parameters to variables.
Mapping rational functions which are not polynomials to polynomials is undefined (i.e. the result depends on the version).

Example:

```
  ring r=0,(x,y,z,a,b,c),dp;
  ideal i=xy2z3a4b5+1,homog(xy2z3a4b5+1,c); i;
↦ i[1]=xy2z3a4b5+1
↦ i[2]=xy2z3a4b5+c15
  ring r1=0,(a,b,x,y,z),lp;
  ideal j=imap(r,i); j;
↦ j[1]=a4b5xy2z3+1
↦ j[2]=a4b5xy2z3
  ring r2=(0,a,b),(x,y,z),ls;
  ideal j=imap(r,i); j;
↦ j[1]=1+(a4b5)*xy2z3
↦ j[2]=(a4b5)*xy2z3
```

See Section 5.1.37 [fetch], page 179; Section 5.1.55 [homog], page 194; Section 4.11 [map], page 104; Section 4.19.1 [qring], page 125; Section 4.19 [ring], page 125.

5.1.58 impart

Syntax: `impart` (number_expression)

Type: number

Purpose: returns the imaginary part of a number in a complex ground field,
returns 0 otherwise.

Example:

```
  ring r=(complex,i),x,dp;
  impart(1+2*i);
↦ 2
```

See Section 5.1.129 [repart], page 248.

5.1.59 indepSet

Syntax: `indepSet` (ideal_expression)

Type: intvec

Purpose: computes a maximal set U of independent variables (in the sense defined in the note below) of the ideal given by a standard basis. If `v` is the result then `v[i]` is 1 if and only if the i-th variable of the ring, `x(i)`, is an independent variable. Hence, the set U consisting of all variables `x(i)` with `v[i]=1` is a maximal independent set.

Note: U is a set of independent variables for I if and only if $I \cap K[U] = (0)$, i.e., eliminating the remaining variables gives (0). U is maximal if dim(I)=#U.

Syntax: `indepSet` (ideal_expression, int_expression)

Type: list

Purpose: computes a list of all maximal independent sets of the leading ideal (if the flag is 0), resp. of all those sets of independent variables of the leading ideal which cannot be enlarged.

Example:

```
  ring r=32003,(x,y,u,v,w),dp;
  ideal I=xyw,yvw,uyw,xv;
  attrib(I,"isSB",1);
  indepSet(I);
↦ 1,1,1,0,0
  eliminate(I,vw);
↦ _[1]=0
  indepSet(I,0);
↦ [1]:
↦    1,1,1,0,0
↦ [2]:
↦    0,1,1,1,0
↦ [3]:
↦    1,0,1,0,1
↦ [4]:
↦    0,0,1,1,1
  indepSet(I,1);
↦ [1]:
↦    1,1,1,0,0
↦ [2]:
↦    0,1,1,1,0
↦ [3]:
↦    1,0,1,0,1
↦ [4]:
↦    0,0,1,1,1
↦ [5]:
↦    0,1,0,0,1
  eliminate(I,xuv);
↦ _[1]=0
```

See Section 4.5 [ideal], page 79; Section 5.1.147 [std], page 266.

5.1.60 insert

Syntax: `insert` (list_expression, expression)
`insert` (list_expression, expression, int_expression)

Type: list

Purpose: inserts a new element (expression) into a list at the beginning, or (if called with 3 arguments) after the given position (the input is not changed).

Example:

```
  list L=1,2;
  insert(L,4,2);
↦ [1]:
↦    1
↦ [2]:
↦    2
↦ [3]:
↦    4
  insert(L,4);
↦ [1]:
↦    4
↦ [2]:
↦    1
↦ [3]:
↦    2
```

See Section 5.1.20 [delete], page 168; Section 4.10 [list], page 102.

5.1.61 interpolation

Syntax: `interpolation` (list, intvec)

Type: ideal

Purpose: `interpolation(l,v)` computes the reduced Groebner basis of the intersection of ideals l[1]^v[1], ..., l[N]^v[N] by applying linear algebra methods.

Assume: Every ideal from the list l must be a maximal ideal of a point and should have the following form: variable_1-coordinate_1, ..., variable_n-coordinate_n, where n is the number of variables in the ring.
The ring should be a polynomial ring over Zp or Q with global ordering.

Example:

```
  ring r=0,(x,y),dp;
  ideal p_1=x,y;
  ideal p_2=x+1,y+1;
  ideal p_3=x+2,y-1;
  ideal p_4=x-1,y+2;
  ideal p_5=x-1,y-3;
  ideal p_6=x,y+3;
  ideal p_7=x+2,y;
  list l=p_1,p_2,p_3,p_4,p_5,p_6,p_7;
  intvec v=2,1,1,1,1,1,1;
  ideal j=interpolation(l,v);
  // generator of degree 3 gives the equation of the unique
  // singular cubic passing
  // through p_1,...,p_7 with singularity at p_1
  j;
↦ j[1]=-4x3-4x2y-2xy2+y3-8x2-4xy+3y2
↦ j[2]=-y4+8x2y+6xy2-2y3+10xy+3y2
↦ j[3]=-xy3+2x2y+xy2+4xy
```

```
↦ j[4]=-2x2y2-2x2y-2xy2+y3-4xy+3y2
  // computes values of generators of j at p_4, results should be 0
  subst(j,x,1,y,-2);
↦ _[1]=0
↦ _[2]=0
↦ _[3]=0
↦ _[4]=0
  // computes values of derivatives d/dx of generators at (0,0)
  subst(diff(j,x),x,0,y,0);
↦ _[1]=0
↦ _[2]=0
↦ _[3]=0
↦ _[4]=0
```

See Section 5.1.23 [diff], page 169; Section 5.1.38 [fglm], page 180; Section 5.1.63 [intersect], page 200; Section 5.1.147 [std], page 266; Section 5.1.150 [subst], page 270.

5.1.62 interred

Syntax: `interred` (ideal_expression)
`interred` (module_expression)

Type: the same as the input type

Purpose: interreduces a set of polynomials/vectors.
Input: $f_1, \ldots, f_n$
Output: $g_1, \ldots, g_s$ with $s \leq n$ and the properties

- $(f_1, \ldots, f_n) = (g_1, \ldots, g_s)$,
- $L(g_i) \neq L(g_j)$ for all $i \neq j$,
- in the case of a global ordering (polynomial ring) and `option(redSB);`:
  $L(g_i)$ does not divide m for all monomials m of $\{g_1, \ldots, g_{i-1}, g_{i+1}, \ldots, g_s\}$,
- in the case of a local ordering (localization of polynomial ring) and `option(redSB);`:
  if $L(g_i)|L(g_j)$ for any $i \neq j$, then $ecart(g_i) > ecart(g_j)$.

Here, $L(g)$ denotes the leading term of g and $ecart(g) := deg(g) - deg(L(g))$.

Example:

```
  ring r=0,(x,y,z),dp;
  ideal i=zx+y3,z+y3,z+xy;
  interred(i);
↦ _[1]=xz-z
↦ _[2]=xy+z
↦ _[3]=y3+xz
  ring R=0,(x,y,z),ds;
  ideal i=zx+y3,z+y3,z+xy;
  interred(i);
↦ _[1]=z+xy
↦ _[2]=xy-y3
↦ _[3]=x2y-y3
```

See Section 4.5 [ideal], page 79; Section 4.13 [module], page 111; Section 5.1.147 [std], page 266.

5.1.63 intersect

Syntax: `intersect` (expression_list of ideal_expression)
`intersect` (expression_list of module_expression)

Type: ideal, resp. module

Purpose: computes the intersection of ideals, resp. modules.

Note: If the option `prot` is enabled then the result the used method (elimination/syzygies) is displayed.
An optional last argument specifies the Groebner base algorith to use. Possible values are `"std"` and `"slimgb"`.

Example:

```
  ring R=0,(x,y),dp;
  ideal i=x;
  ideal j=y;
  intersect(i,j);
↦ _[1]=xy
  ring r=181,(x,y,z),(c,ls);
  ideal id1=maxideal(3);
  ideal id2=x2+xyz,y2-z3y,z3+y5xz;
  ideal id3=intersect(id1,id2,ideal(x,y));
  ideal id4=intersect(id1,id2,"slimgb");
  id3;
↦ id3[1]=yz3+xy6z
↦ id3[2]=yz4-y2z
↦ id3[3]=y2z3-y3
↦ id3[4]=xz3+x2y5z
↦ id3[5]=xyz2+x2z
↦ id3[6]=xy2+x2z2
↦ id3[7]=xy2z+x2y
↦ id3[8]=x2yz+x3
  id4;
↦ id4[1]=xyz2+x2z
↦ id4[2]=xy2z+x2y
↦ id4[3]=x2yz+x3
↦ id4[4]=-yz4+y2z
↦ id4[5]=-y2z3+y3
↦ id4[6]=-xyz3+xy2
↦ id4[7]=z3+xy5z
```

See Section 4.5 [ideal], page 79; Section 4.13 [module], page 111; Section 5.1.108 [option], page 230.

5.1.64 jacob

Syntax: `jacob` (poly_expression)
`jacob` (ideal_expression)
`jacob` (module_expression)

Type: ideal, if the input is a polynomial
matrix, if the input is an ideal
module, if the input is a module

Purpose: computes the Jacobi ideal, resp. Jacobi matrix, generated by all partial derivatives of the input.

Note: In a ring with n variables, jacob of a module or an ideal (considered as matrix with a single a row) or a polynomial (considered as a matrix with a single entry) is the matrix consisting of horizontally concatenated blocks (in this order): diff(MT,var(1)), ... , diff(MT,var(n)), where MT is the transposed input argument considered as a matrix.

Example:

```
  ring R;
  poly f = x2yz + xy3z + xyz5;
  ideal i = jacob(f); i;
↦ i[1]=yz5+y3z+2xyz
↦ i[2]=xz5+3xy2z+x2z
↦ i[3]=5xyz4+xy3+x2y
  matrix m = jacob(i);
  print(m);
↦ 2yz,        z5+3y2z+2xz, 5yz4+y3+2xy,
↦ z5+3y2z+2xz,6xyz,        5xz4+3xy2+x2,
↦ 5yz4+y3+2xy,5xz4+3xy2+x2,20xyz3
  print(jacob(m));
↦ 0, 2z,        2y,        2z,        6yz,5z4+3y2+2x,2y,        5z4+3y2+2x,\
   20yz3,
↦ 2z,6yz,       5z4+3y2+2x,6yz,       6xz,6xy,       5z4+3y2+2x,6xy,       \
   20xz3,
↦ 2y,5z4+3y2+2x,20yz3,     5z4+3y2+2x,6xy,20xz3,     20yz3,     20xz3,     \
   60xyz2
```

See Section 5.1.23 [diff], page 169; Section 4.5 [ideal], page 79; Section 4.13 [module], page 111; Section 5.1.106 [nvars], page 230.

5.1.65 janet

Syntax: `janet` (ideal_expression)
`janet` (ideal_expression , int_expression)

Type: ideal

Purpose: computes the Janet basis of the given ideal, resp. the standard basis if 1 is given as the second argument.

Remark: It works only with global orderings.

Example:

```
  ring r=0,(x,y,z),dp;
  ideal i=x*y*z-1,x+y+z,x*y+x*z+y*z; // cyclic 3
  janet(i);
↦ Length of Janet basis: 4
↦ _[1]=x+y+z
↦ _[2]=y2+yz+z2
↦ _[3]=z3-1
↦ _[4]=yz3-y
```

See [groebner], page 721; Section 4.5 [ideal], page 79; Section 5.1.147 [std], page 266.

5.1.66 jet

Syntax: `jet` (poly_expression, int_expression)
`jet` (vector_expression, int_expression)
`jet` (ideal_expression, int_expression)
`jet` (module_expression, int_expression)
`jet` (poly_expression, int_expression, intvec_expression)
`jet` (vector_expression, int_expression, intvec_expression)
`jet` (ideal_expression, int_expression, intvec_expression)
`jet` (module_expression, int_expression, intvec_expression)
`jet` (poly_expression, poly_expression, int_expression)
`jet` (vector_expression, poly_expression, int_expression)
`jet` (ideal_expression, matrix_expression, int_expression)
`jet` (module_expression, matrix_expression, int_expression)

Type: the same as the type of the first argument

Purpose: deletes from the first argument all terms of degree bigger than the second argument.
If a third argument `w` of type intvec is given, the degree is replaced by the weighted degree defined by `w`.
If a second argument `u` of type poly or matrix is given, the first argument `p` is replaced by `p/u`. In this case, the coeffcient must be from a field.

Example:

```
  ring r=32003,(x,y,z),(c,dp);
  jet(1+x+x2+x3+x4,3);
↦ x3+x2+x+1
  poly f=1+x+x2+xz+y2+x3+y3+x2y2+z4;
  jet(f,3);
↦ x3+y3+x2+y2+xz+x+1
  intvec iv=2,1,1;
  jet(f,3,iv);
↦ y3+y2+xz+x+1
  // the part of f with (total) degree >3:
  f-jet(f,3);
↦ x2y2+z4
  // the homogeneous part of f of degree 2:
  jet(f,2)-jet(f,1);
↦ x2+y2+xz
  // the part of maximal degree:
  jet(f,deg(f))-jet(f,deg(f)-1);
↦ x2y2+z4
  // the absolute term of f:
  jet(f,0);
↦ 1
  // now for other types:
  ideal i=f,x,f*f;
  jet(i,2);
↦ _[1]=x2+y2+xz+x+1
↦ _[2]=x
↦ _[3]=3x2+2y2+2xz+2x+1
  vector v=[f,1,x];
  jet(v,1);
↦ [x+1,1,x]
```

```
  jet(v,0);
↦ [1,1]
  v=[f,1,0];
  module m=v,v,[1,x2,z3,0,1];
  jet(m,2);
↦ _[1]=[x2+y2+xz+x+1,1]
↦ _[2]=[x2+y2+xz+x+1,1]
↦ _[3]=[1,x2,0,0,1]
  ring rs=0,x,ds;
  // 1/(1+x) till degree 5
  jet(1,1+x,5);
↦ 1-x+x2-x3+x4-x5
```

See Section 5.1.18 [deg], page 167; Section 4.5 [ideal], page 79; Section 4.6 [int], page 83; Section 4.8 [intvec], page 92; Section 4.13 [module], page 111; Section 4.16 [poly], page 118; Section 4.21 [vector], page 131.

5.1.67 kbase

Syntax: `kbase` (ideal_expression)
`kbase` (module_expression)
`kbase` (ideal_expression, int_expression)
`kbase` (module_expression, int_expression)

Type: the same as the input type of the first argument

Purpose: With one argument: computes a vector space basis (consisting of monomials) of the quotient ring by the ideal, resp. of a free module by the module, in case it is finite dimensional and if the input is a standard basis with respect to the ring ordering.
Note that, if the input is not a standard basis, the leading terms of the input are used and the result may have no meaning.
With two arguments: computes the part of a vector space basis of the respective quotient with degree of the monomials equal to the second argument. Here, the quotient does not need to be finite dimensional. If an attribute`isHomog` (of type `intvec`) is present, it is used as module weight.

Example:

```
  ring r=32003,(x,y,z),ds;
  ideal i=x2,y2,z;
  kbase(std(i));
↦ _[1]=xy
↦ _[2]=y
↦ _[3]=x
↦ _[4]=1
  i=x2,y3,xyz;  // quotient not finite dimensional
  kbase(std(i),2);
↦ _[1]=z2
↦ _[2]=yz
↦ _[3]=xz
↦ _[4]=y2
↦ _[5]=xy
```

See Section 4.5 [ideal], page 79; Section 4.13 [module], page 111; Section 5.1.163 [vdim], page 280.

5.1.68 kernel

Syntax: `kernel` (ring_name, map_name)
`preimage` (ring_name, ideal_expression)

Type: ideal

Purpose: returns the kernel of a given map.
The second argument has to be a map from the basering to the given ring (or an ideal defining such a map).

Example:

```
  ring r1=32003,(x,y,z,w),lp;
  ring r=32003,(x,y,z),dp;
  ideal i=x,y,z;
  map f=r1,i;
  setring r1;
  // the kernel of f
  kernel(r,f);
↦ _[1]=w
```

See [alg_kernel], page 742; [hom_kernel], page 746; Section 4.5 [ideal], page 79; Section 4.11 [map], page 104; Section 5.1.92 [modulo], page 221; Section 5.1.114 [preimage], page 236; Section 4.19 [ring], page 125.

5.1.69 kill

Syntax: `kill` name
`kill` list_of_names

Type: none

Purpose: deletes objects.

Example:

```
  int i=3;
  ring r=0,x,dp;
  poly p;
  listvar();
↦ // r                          [0]  *ring
↦ //      p                          [0]  poly
↦ // i                          [0]  int 3
  kill i,r;
  // the variable `i` does not exist any more
  i;
↦    ? `i` is undefined
↦    ? error occurred in or before ./examples/kill.sing line 7: `  i;`
  listvar();
```

See Section 5.1.17 [defined], page 166; Section D.2.3 [general_lib], page 726.

5.1.70 killattrib

Syntax: `killattrib` (name)
`killattrib` (name, string_expression)

Type: none

Purpose: deletes all attributes respective the attribute given as the second argument.

Example:

```
  ring r=32003,(x,y),lp;
  ideal i=maxideal(1);
  attrib(i,"isSB",1);
  attrib(i);
↦ attr:isSB, type int
  killattrib(i,"isSB");
  attrib(i);
↦ no attributes
  attrib(i,"isSB",1);
  killattrib(i);
  attrib(i);
↦ no attributes
```

See Section 5.1.2 [attrib], page 154; Section 5.1.108 [option], page 230.

5.1.71 koszul

Syntax: `koszul` (int_expression, int_expression)
`koszul` (int_expression, ideal_expression)
`koszul` (int_expression, int_expression, ideal_expression)

Type: matrix

Purpose: `koszul(d,n)` computes a matrix of the Koszul relations of degree d of the first n ring variables.

`koszul(d,id)` computes a matrix of the Koszul relations of degree d of the generators of the ideal `id`.

`koszul(d,n,id)` computes a matrix of the Koszul relations of degree d of the first n generators of the ideal `id`.

Note: `koszul(1,id)`, `koszul(2,id)`, ... form a complex, that is, the product of the matrices `koszul(i,id)` and `koszul(i+1,id)` equals zero.

Example:

```
  ring r=32003,(x,y,z),dp;
  print(koszul(2,3));
↦ -y,-z,0,
↦ x, 0, -z,
↦ 0, x, y
  ideal I=xz2+yz2+z3,xyz+y2z+yz2,xy2+y3+y2z;
  print(koszul(1,I));
↦ xz2+yz2+z3,xyz+y2z+yz2,xy2+y3+y2z
  print(koszul(2,I));
↦ -xyz-y2z-yz2,-xy2-y3-y2z,0,
↦ xz2+yz2+z3,  0,          -xy2-y3-y2z,
↦ 0,           xz2+yz2+z3, xyz+y2z+yz2
  print(koszul(2,I)*koszul(3,I));
↦ 0,
↦ 0,
↦ 0
```

See Section 4.6 [int], page 83; Section 4.12 [matrix], page 107.

5.1.72 laguerre

Syntax: `laguerre` (poly_expression, int_expression, int_expression)

Type: list

Purpose: computes all complex roots of a univariate polynomial using Laguerre's algorithm. The second argument defines the precision of the fractional part if the ground field is the field of rational numbers, otherwise it will be ignored. The third argument (can be 0, 1 or 2) gives the number of extra runs for Laguerre's algorithm (with corrupted roots), leading to better results.

Note: If the ground field is the field of complex numbers, the elements of the list are of type number, otherwise of type string.

Example:

```
ring rs1=0,(x,y),lp;
poly f=15x5+x3+x2-10;
laguerre(f,10,2);
↦ [1]:
↦    0.8924637479
↦ [2]:
↦    (-0.7392783383+I*0.5355190078)
↦ [3]:
↦    (-0.7392783383-I*0.5355190078)
↦ [4]:
↦    (0.2930464644-I*0.9003002396)
↦ [5]:
↦    (0.2930464644+I*0.9003002396)
```

5.1.73 lead

Syntax: `lead` (poly_expression)
`lead` (vector_expression)
`lead` (ideal_expression)
`lead` (module_expression)

Type: the same as the input type

Purpose: returns the leading (or initial) term(s) of a polynomial, a vector, resp. of the generators of an ideal or module with respect to the monomial ordering.

Note: `IN` may be used instead of `lead`.

Example:

```
  ring r=32003,(x,y,z),(c,ds);
  poly f=2x2+3y+4z3;
  vector v=[2x10,f];
  ideal i=f,z;
  module m=v,[0,0,2+x];
  lead(f);
↦ 3y
  lead(v);
↦ [2x10]
  lead(i);
↦ _[1]=3y
```

```
↦ _[2]=z
  lead(m);
↦ _[1]=[2x10]
↦ _[2]=[0,0,2]
  lead(0);
↦ 0
```

See Section 4.5 [ideal], page 79; Section 5.1.74 [leadcoef], page 207; Section 5.1.75 [leadexp], page 207; Section 5.1.76 [leadmonom], page 208; Section 4.13 [module], page 111; Section 4.16 [poly], page 118; Section 4.21 [vector], page 131.

5.1.74 leadcoef

Syntax: `leadcoef` (poly_expression)
`leadcoef` (vector_expression)

Type: number

Purpose: returns the leading (or initial) coefficient of a polynomial or a vector with respect to the monomial ordering.

Example:

```
  ring r=32003,(x,y,z),(c,ds);
  poly f=x2+y+z3;
  vector v=[2*x^10,f];
  leadcoef(f);
↦ 1
  leadcoef(v);
↦ 2
  leadcoef(0);
↦ 0
```

See Section 5.1.73 [lead], page 206; Section 5.1.75 [leadexp], page 207; Section 5.1.76 [leadmonom], page 208; Section 4.16 [poly], page 118; Section 4.21 [vector], page 131.

5.1.75 leadexp

Syntax: `leadexp` (poly_expression)
`leadexp` (vector_expression)

Type: intvec

Purpose: returns the exponent vector of the leading monomial of a polynomial or a vector. In the case of a vector the last component is the index in the vector. (The inverse to `monomial`.)

Example:

```
  ring r=32003,(x,y,z),(c,ds);
  poly f=x2+y+z3;
  vector v=[2*x^10,f];
  leadexp(f);
↦ 0,1,0
  leadexp(v);
↦ 10,0,0,1
  leadexp(0);
↦ 0,0,0
```

See Section 4.8 [intvec], page 92; Section 5.1.73 [lead], page 206; Section 5.1.74 [leadcoef], page 207; Section 5.1.76 [leadmonom], page 208; Section 5.1.94 [monomial], page 222; Section 4.16 [poly], page 118; Section 4.21 [vector], page 131.

5.1.76 leadmonom

Syntax: `leadmonom` (poly_expression)
`leadmonom` (vector_expression)

Type: the same as the input type

Purpose: returns the leading monomial of a polynomial or a vector as a polynomial or vector whose coefficient is one.

Example:

```
  ring r=32003,(x,y,z),(c,ds);
  poly f=2x2+3y+4z3;
  vector v=[0,2x10,f];
  leadmonom(f);
↦ y
  leadmonom(v);
↦ [0,x10]
  leadmonom(0);
↦ 0
```

See Section 4.8 [intvec], page 92; Section 5.1.73 [lead], page 206; Section 5.1.74 [leadcoef], page 207; Section 5.1.75 [leadexp], page 207; Section 4.16 [poly], page 118; Section 4.21 [vector], page 131.

5.1.77 LIB

Syntax: `LIB` string_expression`;`

Type: none

Purpose: reads a library of procedures from a file. In contrast to the command `load`, the procedures from the library are added to the package `Top` as well as the package corresponding to the library. If the given filename does not start with `.` or `/` and cannot be located in the current directory, each directory contained in the library `SearchPath` is searched for file of this name. See Section 3.8.11 [Loading a library], page 67, for more info on `SearchPath`.

Note on standard.lib:
Unless Singular is started with the `--no-stdlib` option, the library `standard.lib` is automatically loaded at start-up time.

Example:

```
  option(loadLib); // show loading of libraries

                   // the names of the procedures of inout.lib
  LIB "inout.lib"; // are now known to Singular
↦ // ** loaded inout.lib (4.1.1.0,Dec_2017)
```

See Section 3.1.6 [Command line options], page 19; Section 2.3.3 [Procedures and libraries], page 10; Appendix D [SINGULAR libraries], page 721; Section 5.2.12 [load], page 293; Section 4.15 [package], page 118; Section 4.17 [proc], page 122; Section D.1 [standard_lib], page 721; Section 4.20 [string], page 128; Section 5.1.151 [system], page 270.

5.1.78 lift

Syntax: `lift` (ideal_expression, subideal_expression)
`lift` (module_expression, submodule_expression)
`lift` (ideal_expression, subideal_expression, matrix_name)
`lift` (module_expression, submodule_expression, matrix_name)
`lift` (ideal_expression, subideal_expression, matrix_name, string_expression)
`lift` (module_expression, submodule_expression, matrix_name, string_expression)

Type: matrix

Purpose: computes the transformation matrix which expresses the generators of a submodule in terms of the generators of a module. Depending on which algorithm is used, modules are represented by a standard basis, or not.
More precisely, if `m` is the module (or ideal), `sm` the submodule (or ideal), and `T` the transformation matrix returned by lift, then `matrix(sm)*U = matrix(m)*T` and `module(sm*U) = module(matrix(m)*T)` (resp. `ideal(sm) = ideal(matrix(m)*T)`), where `U` is a diagonal matrix of units.
`U` is always the identity if the basering is a polynomial ring (not power series ring). `U` is stored in the optional third argument.

Note: Gives a warning if `sm` is not a submodule.
An optional 4th argument specifies the Groebner base algorith to use. Possible values are `"std"` and `"slimgb"`.

Example:

```
  ring r=32003,(x,y,z),(dp,C);
  ideal m=3x2+yz,7y6+2x2y+5xz;
  poly f=y7+x3+xyz+z2;
  ideal i=jacob(f);
  matrix T=lift(i,m);
  matrix(m)-matrix(i)*T;
↦ _[1,1]=0
↦ _[1,2]=0
```

See Section 5.1.25 [division], page 171; Section 4.5 [ideal], page 79; Section 5.1.79 [liftstd], page 209; Section 4.13 [module], page 111; Section 5.1.147 [std], page 266; Section 5.1.152 [syz], page 275.

5.1.79 liftstd

Syntax: `liftstd` (ideal_expression, matrix_name)
`liftstd` (module_expression, matrix_name)
`liftstd` (ideal_expression, matrix_name, module_name)
`liftstd` (module_expression, matrix_name, module_name)
`liftstd` (ideal_expression, matrix_name, module_name, string_expression)
`liftstd` (module_expression, matrix_name, module_name, string_expression)

Type: ideal or module

Purpose: returns a standard basis of an ideal or module and the transformation matrix from the given ideal, resp. module, to the standard basis.
That is, if `m` is the ideal or module, `sm` the standard basis returned by `liftstd`, and `T` the transformation matrix then `matrix(sm)=matrix(m)*T` and `sm=ideal(matrix(m)*T)`, resp. `sm=module(matrix(m)*T)`.
In an optional third argument the syzygy module will be returned. if

working in a quotient ring, then `matrix(sm)=reduce(matrix(m)*T,0)` and `sm=reduce(ideal(matrix(m)*T),0)`.
An optional 4th argument specifies the Groebner base algorith to use. Possible values are `"std"` and `"slimgb"`.

Example:

```
  ring R=0,(x,y,z),dp;
  poly f=x3+y7+z2+xyz;
  ideal i=jacob(f);
  matrix T;
  ideal sm=liftstd(i,T);
  sm;
↦ sm[1]=xy+2z
↦ sm[2]=3x2+yz
↦ sm[3]=yz2+3048192z3
↦ sm[4]=3024xz2-yz2
↦ sm[5]=y2z-6xz
↦ sm[6]=3097158156288z4+2016z3
↦ sm[7]=7y6+xz
  print(T);
↦ 0,1,T[1,3],   T[1,4],y,  T[1,6],0,
↦ 0,0,-3x+3024z,3x,    0,  T[2,6],1,
↦ 1,0,T[3,3],   T[3,4],-3x,T[3,6],0
  matrix(sm)-matrix(i)*T;
↦ _[1,1]=0
↦ _[1,2]=0
↦ _[1,3]=0
↦ _[1,4]=0
↦ _[1,5]=0
↦ _[1,6]=0
↦ _[1,7]=0
  module s;
  sm=liftstd(i,T,s);
  print(s);
↦ -xy-2z,7y6+xz, -7056y6-1008xz,
↦ 0,     -3x2-yz,3024x2-xy+1008yz-2z,
↦ 3x2+yz,0,      7y6+xz
```

See Section 5.1.25 [division], page 171; Section 4.5 [ideal], page 79; Section 5.1.78 [lift], page 209; Section 4.12 [matrix], page 107; Section 5.1.108 [option], page 230; Section 4.19 [ring], page 125; Section 5.1.147 [std], page 266; Section 5.1.152 [syz], page 275.

5.1.80 listvar

Syntax: `listvar` ([package])
`listvar` ([package,] type)
`listvar` ([package,] ring_name)
`listvar` ([package,] name)
`listvar` ([package,] `all`)

Type: none

Purpose: lists all (user-)defined names:

- `listvar()`: all currently visible names except procedures in the current namespace,
- `listvar(`type`)`: all currently visible names of the given type,
- `listvar(`ring_name`)`: all names which belong to the given ring,
- `listvar(`name`)`: the object with the given name,
- `listvar(all)`: all names except procedures in the current and `Top` namespace.

The current basering is marked with a `*`. The nesting level of variables in procedures is shown in square brackets.

package can be `Current`, `Top` or any other identifier of type package.

Example:

```
  proc t1 { }
  proc t2 { }
  ring s;
  poly ss;
  ring r;
  poly f=x+y+z;
  int i=7;
  ideal I=f,x,y;
  listvar();
↦ // i                              [0]  int 7
↦ // r                              [0]  *ring
↦ //      I                              [0]  ideal, 3 generator(s)
↦ //      f                              [0]  poly
↦ // s                              [0]  ring
  listvar(r);
↦ // r                              [0]  *ring
↦ // I                              [0]  ideal, 3 generator(s)
↦ // f                              [0]  poly
  listvar(t1);
↦ // t1                             [0]  proc
  listvar(proc);
↦ // t2                             [0]  proc
↦ // t1                             [0]  proc
↦ // mathicgb_prOrder               [0]  proc from singmathic.so (C)
↦ // mathicgb                       [0]  proc from singmathic.so (C)
↦ // min                            [0]  proc from standard.lib
↦ // max                            [0]  proc from standard.lib
↦ // datetime                       [0]  proc from standard.lib
↦ // weightKB                       [0]  proc from standard.lib
↦ // fprintf                        [0]  proc from standard.lib
↦ // printf                         [0]  proc from standard.lib
↦ // sprintf                        [0]  proc from standard.lib
↦ // quotient4                      [0]  proc from standard.lib
↦ // quotient5                      [0]  proc from standard.lib
↦ // quotient3                      [0]  proc from standard.lib
↦ // quotient2                      [0]  proc from standard.lib
↦ // quotient1                      [0]  proc from standard.lib
↦ // quot                           [0]  proc from standard.lib
↦ // res                            [0]  proc from standard.lib
↦ // groebner                       [0]  proc from standard.lib
```

```
↦ // qslimgb                          [0]  proc from standard.lib
↦ // hilbRing                         [0]  proc from standard.lib
↦ // par2varRing                      [0]  proc from standard.lib
↦ // quotientList                     [0]  proc from standard.lib
↦ // stdhilb                          [0]  proc from standard.lib
↦ // stdfglm                          [0]  proc from standard.lib
↦ // flintZ                           [0]  proc from kernel (C)
↦ // Float                            [0]  proc from kernel (C)
↦ // crossprod                        [0]  proc from kernel (C)
  LIB "poly.lib";
  listvar(Poly);
↦ // Poly                             [0]  package Poly (S,poly.lib)
↦ // ::newtonDiag                     [0]  proc from poly.lib
↦ // ::subrInterred                   [0]  proc from poly.lib
↦ // ::id2mod                         [0]  proc from poly.lib
↦ // ::mod2id                         [0]  proc from poly.lib
↦ // ::content                        [0]  proc from poly.lib
↦ // ::lcm                            [0]  proc from poly.lib
↦ // ::rad_con                        [0]  proc from poly.lib
↦ // ::normalize                      [0]  proc from poly.lib
↦ // ::mindeg1                        [0]  proc from poly.lib
↦ // ::mindeg                         [0]  proc from poly.lib
↦ // ::maxdeg1                        [0]  proc from poly.lib
↦ // ::maxdeg                         [0]  proc from poly.lib
↦ // ::maxcoef                        [0]  proc from poly.lib
↦ // ::is_zero                        [0]  proc from poly.lib
↦ // ::freerank                       [0]  proc from poly.lib
↦ // ::kat_var                        [0]  proc from poly.lib
↦ // ::katsura                        [0]  proc from poly.lib
↦ // ::elemSymmId                     [0]  proc from poly.lib
↦ // ::elemSymmPoly                   [0]  proc from poly.lib
↦ // ::cyclic                         [0]  proc from poly.lib
↦ // ::substitute                     [0]  proc from poly.lib
↦ // ::hilbPoly                       [0]  proc from poly.lib
↦ // ::bino                           [0]  proc from poly.lib (static)
```

See Section 3.5.3 [Names], page 44; Section 3.7.4 [Names in procedures], page 55; Section 5.1.17 [defined], page 166; Section 5.1.100 [names], page 225; Section 4.15 [package], page 118; Section 5.1.155 [type], page 277.

5.1.81 lres

Syntax: `lres (` ideal_expression`,` int_expression `)`

Type: resolution

Purpose: computes a free resolution of an ideal using LaScala's algorithm.

More precisely, let R be the basering and I be the given ideal. Then `lres` computes a minimal free resolution of R/I

$$... \longrightarrow F_2 \xrightarrow{A_2} F_1 \xrightarrow{A_1} R \longrightarrow R/I \longrightarrow 0.$$

If the int_expression k is not zero then the computation stops after k steps and returns a list of modules $M_i =$ `module`(A_i), i=1..k.

`list L=lres(I,0);` returns a list L of n modules (where n is the number of variables of the basering) such that `L[i]` $= M_i$ in the above notation.

Note: The ideal_expression has to be homogeneous.
Accessing single elements of a resolution may require that some partial computations have to be finished and may therefore take some time.

Example:

```
  ring r=0,(x,y,z),dp;
  ideal I=xz,yz,x3-y3;
  def L=lres(I,0);
  print(betti(L),"betti");
↦            0     1     2
↦ ------------------------
↦     0:     1     -     -
↦     1:     -     2     1
↦     2:     -     1     1
↦ ------------------------
↦ total:     1     3     2
↦
  L[2];     // the first syzygy module of r/I
↦ _[1]=-x*gen(1)+y*gen(2)
↦ _[2]=-x2*gen(2)+y2*gen(1)+z*gen(3)
```

See Section 5.1.4 [betti], page 157; Section 5.1.46 [fres], page 185; Section 5.1.56 [hres], page 195; Section 4.5 [ideal], page 79; Section 4.6 [int], page 83; Section 5.1.91 [minres], page 220; Section 4.13 [module], page 111; Section 5.1.96 [mres], page 223; [res], page 721; Section 5.1.145 [sres], page 264.

5.1.82 ludecomp

Syntax: `ludecomp (` matrix_expression `)`

Type: list

Purpose: Computes the LU-decomposition of an (m x n) matrix.

The matrix, A say, must consist of numbers, only. This means that when the basering represents some $K[x_1, x_2, \ldots, x_r]$, then all entries of A must come from the ground field K.
The LU-decomposition of A is a triple of matrices P, L, and U such that
- P * A = L * U,
- P is an (m x m) permutation matrix, i.e., its rows/columns form the standard basis of K^m,
- L is an (m x m) matrix in lower triangular form with all diagonal entries equal to 1, and
- U is an (m x n) matrix in upper row echelon form.
From these conditions, it easily follows that also A = P * L * U holds, since P is self-inverse.

`list L=ludecomp(A);` fills a list L with the three above entries P, L, and U.

Example:

```
  ring r=0,(x),dp;
  matrix A[3][4]=1,2,3,4,1,1,1,1,2,2,1,1;
  list plu = ludecomp(A);
  print(plu[3]);                  // the matrix U of the decomposition
```

```
↦ 1,2, 3, 4,
↦ 0,-1,-2,-3,
↦ 0,0, -1,-1
  print(plu[1]*A-plu[2]*plu[3]);   // should be the zero matrix
↦ 0,0,0,0,
↦ 0,0,0,0,
↦ 0,0,0,0
```

See Section 5.1.83 [luinverse], page 214.

5.1.83 luinverse

Syntax: `luinverse` (matrix_expression)

Type: matrix

Syntax: `luinverse` (matrix_expression, matrix_expression, matrix_expression)

Type: matrix

Purpose: Computes the inverse of a matrix A, if A is invertible.

The matrix A must be given either directly, or by its LU-decomposition. In the latter case, three matrices P, L, and U are expected, in this order, which satisfy
- P * A = L * U,
- P is an (m x m) permutation matrix, i.e., its rows/columns form the standard basis of K^m,
- L is an (m x m) matrix in lower triangular form with all diagonal entries equal to 1, and
- U is an (m x m) matrix in upper row echelon form.
Then, the inverse of A exists if and only if U is invertible, and one has $A^{-1} = U^{-1} \cdot L^{-1} \cdot P$, since P is self-inverse.
In the case of A being given directly, `luinverse` first computes its LU-decomposition, and then proceeds as in the case when P, L, and U are provided.

`list L=luinverse(A);` fills the list L with either one entry = 0 (signaling that A is not invertible), or with the two entries $1, A^{-1}$. Thus, in either case the user may first check the condition `L[1]==1` to find out whether A is invertible.

Note: The method will give a warning for any non-quadratic matrix A.

Example:

```
  ring r=0,(x),dp;
  matrix A[3][3]=1,2,3,1,1,1,2,2,1;
  list L = luinverse(A);
  if (L[1] == 1)
  {
    print(L[2]);
    "----- next should be the (3 x 3)-unit matrix:";
    print(A*L[2]);
  }
↦ -1,4, -1,
↦ 1, -5,2,
↦ 0, 2, -1
↦ ----- next should be the (3 x 3)-unit matrix:
↦ 1,0,0,
↦ 0,1,0,
↦ 0,0,1
```

See Section 5.1.82 [ludecomp], page 213.

5.1.84 lusolve

Syntax: `lusolve (` matrix_expression`,` matrix_expression`,` matrix_expression`,` matrix_expression `)`

Type: matrix

Purpose: Computes all solutions of a linear equation system A*x = b, if solvable
The (m x n matrix A must be given by its LU-decomposition, that is, by three matrices P, L, and U, in this order, which satisfy
- P * A = L * U,
- P is an (m x m) permutation matrix, i.e., its rows/columns form the standard basis of K^m,
- L is an (m x m) matrix in lower triangular form with all diagonal entries equal to 1, and
- U is an (m x n) matrix in upper row echelon form.
The fourth argument, b, is expected to be an (m x 1) matrix.

`list Q=lusolve(P,L,U,b);` fills the list Q with either one entry = 0 (signaling that A*x=b has no solution), or with the three entries 1, x, H, where x is any (n x 1) solution of the given linear system, and H is a matrix the columns of which span the solution space of the homogeneous linear system. (I.e., `ncols(H)` is the dimension of the solution space.)
If there is exactly one solution, then H is the 1x1 matrix with entry zero.

Note: The method will give a warning if the matrices violate the above conditions regarding row and column numbers, or if the number of rows of the vector b does not equal m.
The method expects matrices with entries coming from the ground field of the given polynomial ring, only.

Example:

```
  ring r=0,(x),dp;
  matrix A[4][4]=1,1,1,0,1,2,3,1,1,3,5,2,1,4,7,3;
  matrix b[4][1]=2,5,8,11;
  list L=ludecomp(A);
  list Q=lusolve(L[1],L[2],L[3],b);
  if (Q[1] == 1)
  {
    "one solution:";
    print(Q[2]);
    "check whether result is correct (iff next is zero vector):";
    print(A*Q[2]-b);
    if ((nrows(Q[3])==1) and (ncols(Q[3])==1) and (Q[3][1,1]==0))
    { "printed solution is the only solution to given linear system" }
    else
    {
      "homogeneous solution space is spanned by columns of:";
      print(Q[3]);
    }
  }
↦ one solution:
↦ -1,
```

```
↦ 3,
↦ 0,
↦ 0
↦ check whether result is correct (iff next is zero vector):
↦ 0,
↦ 0,
↦ 0,
↦ 0
↦ homogeneous solution space is spanned by columns of:
↦ -1,-1,
↦ 1, 2,
↦ 0, -1,
↦ -1,0
```

See Section 5.1.82 [ludecomp], page 213.

5.1.85 max

Procedure from library `standard.lib` (see Section D.1 [standard_lib], page 721).

Syntax: max (i_1, ..., i_k)

Type: same as type of i_1, ..., i_k resp.

Purpose: returns the maximum for any arguments of a type for which '>' is defined

Example:

```
  // biggest int
max(2,3);
↦ 3
max(1,4,3);
↦ 4
// lexicographically biggest intvec
max(intvec(1,2),intvec(0,1),intvec(1,1));
↦ 1,2
// polynopmial with biggest leading monomial
ring r = 0,x,dp;
max(x+1,x2+x);
↦ x2+x
```

See also: [min], page 721.

5.1.86 maxideal

Syntax: `maxideal` (int_expression)

Type: ideal

Purpose: returns the power given by int_expression of the maximal ideal generated by all ring variables (`maxideal(i)=1` for `i<=0`).

Example:

```
  ring r=32003,(x,y,z),dp;
  maxideal(2);
↦ _[1]=z2
↦ _[2]=yz
```

```
↦ _[3]=y2
↦ _[4]=xz
↦ _[5]=xy
↦ _[6]=x2
```

See Section 4.5 [ideal], page 79; Section 4.19 [ring], page 125.

5.1.87 memory

Syntax: `memory` (int_expression)

Type: bigint

Purpose: returns statistics concerning the memory management:

- `memory(0)` is the number of active (used) bytes,
- `memory(1)` is the number of bytes allocated from the operating system,
- `memory(2)` is the maximal number of bytes ever allocated from the operating system during the current SINGULAR session.

Note: To monitor the memory usage during ongoing computations the option `mem` should be set (using the command `option(mem);`, see also Section 5.1.108 [option], page 230).

Example:

```
  ring r=0,(x(1..500)),dp;
  poly p=(x(1)+x(500))^50;
  proc ReportMemoryUsage()
  {  "Memory currently used by SINGULAR     :",memory(0),"Byte (",
     int(memory(0) div 1024), "KByte)" +newline+
     "Memory currently allocated from system:",memory(1), "Byte (",
     int(memory(1) div 1024), "KByte)";
     "Maximal memory allocated from system  :",memory(2), "Byte (",
     int(memory(2) div 1024), "KByte)";
  }
  ReportMemoryUsage();
↦ Memory currently used by SINGULAR     : 153408 Byte ( 149 KByte)
↦ Memory currently allocated from system: 2228224 Byte ( 2176 KByte)
↦ Maximal memory allocated from system  : 2232320 Byte ( 2180 KByte)
  kill p;
  ReportMemoryUsage(); // less memory used: p killed
↦ Memory currently used by SINGULAR     : 82848 Byte ( 81 KByte)
↦ Memory currently allocated from system: 2228224 Byte ( 2176 KByte)
↦ Maximal memory allocated from system  : 2232320 Byte ( 2180 KByte)
  kill r;
  ReportMemoryUsage(); // even less memory: r killed
↦ Memory currently used by SINGULAR     : 70648 Byte ( 69 KByte)
↦ Memory currently allocated from system: 2228224 Byte ( 2176 KByte)
↦ Maximal memory allocated from system  : 2232320 Byte ( 2180 KByte)
```

See Section 5.1.108 [option], page 230; Section 5.1.151 [system], page 270.

5.1.88 min

Procedure from library `standard.lib` (see Section D.1 [standard_lib], page 721).

Syntax: max (i_1, ..., i_k)

Type: same as type of i_1, ..., i_k resp.

Purpose: returns the maximum for any arguments of a type for which '>' is defined

Example:

```
  // biggest int
max(2,3);
↦ 3
max(1,4,3);
↦ 4
// lexicographically biggest intvec
max(intvec(1,2),intvec(0,1),intvec(1,1));
↦ 1,2
// polynopmial with biggest leading monomial
ring r = 0,x,dp;
max(x+1,x2+x);
↦ x2+x
```

See also: [min], page 721.

5.1.89 minbase

Syntax: `minbase` (ideal_expression)
`minbase` (module_expression)

Type: the same as the type of the argument

Purpose: returns a minimal set of generators of an ideal, resp. module, if the input is either homogeneous or if the ordering is local.

Note: this command is not available over coefficient rings.

Example:

```
  ring r=181,(x,y,z),(c,ls);
  ideal id2=x2+xyz,y2-z3y,z3+y5xz;
  ideal id4=maxideal(3)+id2;
  size(id4);
↦ 13
  minbase(id4);
↦ _[1]=x2
↦ _[2]=xyz+x2
↦ _[3]=xz2
↦ _[4]=y2
↦ _[5]=yz2
↦ _[6]=z3
```

See Section 5.1.97 [mstd], page 223.

5.1.90 minor

Syntax: `minor (` matrix_expression M`,` int_expression mSize`,`
`[` ideal_expression I`],`
`[` int_expression k`],`
`[` string_expression algorithm`],`
`[` int_expression cachedP`],`
`[` int_expression cachedM`])`

Type: ideal

Purpose: returns the specified set of (mSize x mSize)-minors (= subdeterminants) of the given matrix M. These minors form the list of generators of the returned ideal.
If the optional ideal I is given, it is assumed to capture a standard basis. In this case, all computations will be performed modulo I.
If k is not given, all minors will be computed. Otherwise, if k `>` 0, the first k non-zero minors will be computed; for k `<` 0, the first `|k|` minors will be computed regardless whether they are zero or not. Here, "first k minors" is with respect to a fixed ordering among all minors. (To understand the ordering, run the below example, type `minor(m,2,i,18);` and inspect the ordering among the returned 18 minors. Note that this ordering is only enforced when some k `<>` 0 is provided. Otherwise, no ordering among the returned minors can be guaranteed. This is due to the fact that in this case, `minor` may call a specially tuned implementation of Bareiss's algorithm.)
If no algorithm is given, a heuristic will pick the best-suited algorithm among Bareiss's algorithm (which is only applicable over integral domains), Laplace's algorithm, and Laplace's algorithm combined with caching of subdeterminantes. In the heuristic setting, cacheP and cacheM must also be absent.
If the argument algorithm is present it must be one of `B/bareiss`, `L/laplace`, and `C/cache`. For, `B/bareiss` and `L/laplace` the optional arguments cacheP and cacheM must again be absent, whereas for `C/cache`, they may be provided: cachedP determines the maximum number of cached subdeterminantes (=polynomials), and cachedM the total number of cached monomials (counted over all cached polynomials). If, for algorithm = `C/cache` cachedP and cachedM are not provided by the user, the values 200 and 100000, respectively, will be used as defaults.

Note: If `mSize` is larger than the given matrix, `minor` returns 0, if `mSize` is smaller than 1, `minor` returns 1.

Example:

```
  ring r=0,(a,b,c,d,e,f,g,h,s,t,u,v),ds;
  matrix m[3][4]=a,b,c,d,e,f,g,h,s,t,u,v;
  print(m);
↦ a,b,c,d,
↦ e,f,g,h,
↦ s,t,u,v
  // let's compute all non-zero minors;
  // here we do not guarantee any ordering:
  minor(m,2);
↦ _[1]=-hu+gv
↦ _[2]=-ht+fv
↦ _[3]=-hs+ev
↦ _[4]=-du+cv
↦ _[5]=-dt+bv
↦ _[6]=-ds+av
↦ _[7]=gt-fu
↦ _[8]=gs-eu
↦ _[9]=ct-bu
↦ _[10]=cs-au
↦ _[11]=-fs+et
↦ _[12]=-bs+at
↦ _[13]=-dg+ch
↦ _[14]=-df+bh
```

```
↦ _[15]=-de+ah
↦ _[16]=cf-bg
↦ _[17]=ce-ag
↦ _[18]=-be+af
  ideal i=a,c; i=std(i);
  // here come the first 4 non-zero minors mod I;
  // this time, a fixed ordering is guaranteed:
  minor(m,2,i,4);
↦ _[1]=-be
↦ _[2]=bg
↦ _[3]=-de
↦ _[4]=-df+bh
  // and here the first 4 minors mod I (possibly zero)
  // using Laplace's algorithm,
  // again, the fixed ordering is guaranteed:
  minor(m,2,i,-4,"Laplace");
↦ _[1]=-be
↦ _[2]=0
↦ _[3]=bg
↦ _[4]=-de
```

See Section 5.1.22 [det], page 169.

5.1.91 minres

Syntax: `minres` (list_expression)

Type: list

Syntax: `minres` (resolution_expression)

Type: resolution

Purpose: minimizes a free resolution of an ideal or module given by the list_expression, resp. resolution_expression.

Example:

```
  ring r1=32003,(x,y),dp;
  ideal i=x5+xy4,x3+x2y+xy2+y3;
  resolution rs=lres(i,0);
  rs;
↦   1      2      1
↦ r1 <--  r1 <--  r1
↦
↦ 0      1      2
↦
  list(rs);
↦ [1]:
↦    _[1]=x3+x2y+xy2+y3
↦    _[2]=xy4
↦ [2]:
↦    _[1]=xy4*gen(1)-x3*gen(2)-x2y*gen(2)-xy2*gen(2)-y3*gen(2)
  minres(rs);
↦   1      2      1
↦ r1 <--  r1 <--  r1
↦
```

```
↦ 0       1       2
↦
  list(rs);
↦ [1]:
↦    _[1]=x3+x2y+xy2+y3
↦    _[2]=xy4
↦ [2]:
↦    _[1]=xy4*gen(1)-x3*gen(2)-x2y*gen(2)-xy2*gen(2)-y3*gen(2)
```

See Section 5.1.46 [fres], page 185; Section 5.1.96 [mres], page 223; [res], page 721; Section 5.1.145 [sres], page 264.

5.1.92 modulo

Syntax: `modulo` (ideal_expression`,` ideal_expression)
`modulo` (module_expression`,` module_expression)

Type: module

Purpose: `modulo(h1,h2)` represents $h_1/(h_1 \cap h_2) \cong (h_1 + h_2)/h_2$ where h_1 and h_2 are considered as submodules of the same free module R^l (l=1 for ideals). Let H_1, resp. H_2, be the matrices of size $l \times k$, resp. $l \times m$, having the generators of h_1, resp. h_2, as columns. Then $h_1/(h_1 \cap h_2) \cong R^k/ker(\overline{H_1})$ where $\overline{H_1} : R^k \to R^l/Im(H_2) = R^l/h_2$ is the induced map.
`modulo(h1,h2)` returns generators of the kernel of this induced map.

Note: If for at least one of `h1` or `h2` the attribute `"isHomog"` is set, `modulo(h1,h2)` also sets the attribute `"isHomog"` (if possible, that is, if the weights are compatible).

Example:

```
  ring r;
  ideal h1=x,y,z;
  ideal h2=x;
  module m=modulo(h1,h2);
  print(m);
↦ 1,0, 0,0,
↦ 0,-z,x,0,
↦ 0,y, 0,x
```

See [hom_kernel], page 746; Section 5.1.152 [syz], page 275.

5.1.93 monitor

Syntax: `monitor` (link_expression)
`monitor` (link_expression`,` string_expression)

Type: none

Purpose: controls the recording of all user input and/or program output into a file. The second argument describes what to log: `"i"` means input, `"o"` means output, `"io"` for both. The default for the second argument is `"i"`.
Each `monitor` command closes a previous monitor file and opens the file given by the first string expression.
`monitor ("")` turns off recording.

Example:

```
monitor("doe.tmp","io"); // log input and output to doe.tmp
ring r;
poly f=x+y+z;
int i=7;
ideal I=f,x,y;
monitor("");             // stop logging:
// doe.tmp contains now all input and output from the example above
```

See Section 4.9.2 [link expressions], page 95.

5.1.94 monomial

Syntax: `monomial` (intvec_expression)

Type: poly resp. vector

Purpose: converts an integer vector to a power product (the inverse to `leadexp`).
Returns a `vector` iff the length of the argument is number of variables `+`1.

Example:

```
  ring r=0,(x,y,z),dp;
  monomial(intvec(2,3));
↦ x2y3
  monomial(intvec(2,3,0,1));
↦ x2y3*gen(1)
  leadexp(monomial(intvec(2,3,0,1)));
↦ 2,3,0,1
```

See Section 4.8 [intvec], page 92; Section 5.1.75 [leadexp], page 207.

5.1.95 mpresmat

Syntax: `mpresmat` (ideal_expression, int_expression)

Type: module

Purpose: computes the multipolynomial resultant matrix of the input system. Uses the sparse resultant matrix method of Gelfand, Kapranov and Zelevinsky (second parameter = 0) or the resultant matrix method of Macaulay (second parameter = 1).

Note: When using the resultant matrix method of Macaulay the input system must be homogeneous. The number of elements in the input system must be the number of variables in the basering plus one.

Example:

```
  ring rsq=(0,s,t,u),(x,y),lp;
  ideal i=s+tx+uy,x2+y2-10,x2+xy+2y2-16;
  module m=mpresmat(i,0);
  print(m);
↦ -16,0,  -10,0,  (s),0,  0,  0,  0,  0,
↦ 0,  -16,0,  -10,(u),(s),0,  0,  0,  0,
↦ 2,  0,  1,  0,  0,  (u),0,  0,  0,  0,
↦ 0,  2,  0,  1,  0,  0,  0,  0,  0,  0,
↦ 0,  0,  0,  0,  (t),0,  -10,(s),0,  -16,
↦ 1,  0,  0,  0,  0,  (t),0,  (u),(s),0,
↦ 0,  1,  0,  0,  0,  0,  1,  0,  (u),2,
```

```
↦ 1,  0,  1,  0,  0,  0,  0,  (t),0,  0,
↦ 0,  1,  0,  1,  0,  0,  0,  0,  (t),1,
↦ 0,  0,  0,  0,  0,  0,  1,  0,  0,  1
```

See Section 5.1.158 [uressolve], page 278.

5.1.96 mres

Syntax: `mres` (ideal_expression`,` int_expression)
`mres` (module_expression`,` int_expression)

Type: resolution

Purpose: computes a minimal free resolution of an ideal or module M with the standard basis method. More precisely, let A=`matrix`(M), then `mres` computes a free resolution of $coker(A) = F_0/M$

$$... \longrightarrow F_2 \xrightarrow{A_2} F_1 \xrightarrow{A_1} F_0 \longrightarrow F_0/M \longrightarrow 0,$$

where the columns of the matrix A_1 are a minimal set of generators of M if the basering is local or if M is homogeneous. If the int expression k is not zero, then the computation stops after k steps and returns a list of modules $M_i =$ `module`(A_i), i=1...k.
`mres(M,0)` returns a resolution consisting of at most n+2 modules, where n is the number of variables of the basering. Let `list L=mres(M,0);` then `L[1]` consists of a minimal set of generators of the input, `L[2]` consists of a minimal set of generators for the first syzygy module of `L[1]`, etc., until `L[p+1]`, such that `L[i]` $\neq 0$ for $i \leq p$, but `L[p+1]`, the first syzygy module of `L[p]`, is 0 (if the basering is not a qring).

Note: Accessing single elements of a resolution may require some partial computations to be finished and may therefore take some time.

Example:

```
  ring r=31991,(t,x,y,z,w),ls;
  ideal M=t2x2+tx2y+x2yz,t2y2+ty2z+y2zw,
          t2z2+tz2w+xz2w,t2w2+txw2+xyw2;
  resolution L=mres(M,0);
  L;
↦  1      4      15      18      7      1
↦ r <--  r <--  r <--   r <--   r <--  r
↦
↦ 0      1      2       3       4      5
↦
  // projective dimension of M is 5
```

See Section 5.1.46 [fres], page 185; Section 5.1.56 [hres], page 195; Section 4.5 [ideal], page 79; Section 5.1.81 [lres], page 212; Section 4.13 [module], page 111; [res], page 721; Section 5.1.145 [sres], page 264.

5.1.97 mstd

Syntax: `mstd` (ideal_expression)
`mstd` (module_expression)

Type: list

Purpose: returns a list whose first entry is a standard basis for the ideal, resp. module, whose second entry is a generating set for the ideal, resp. module. If the coefficient ring is a field and either the ideal/module is homogeneous or the ordering is local, this second entry is a minimal generating set.

Example:

```
  ring r=0,(x,y,z,t),dp;
  poly f=x3+y4+z6+xyz;
  ideal j=jacob(f),f;
  j=homog(j,t);j;
↦ j[1]=3x2+yz
↦ j[2]=4y3+xzt
↦ j[3]=6z5+xyt3
↦ j[4]=0
↦ j[5]=z6+y4t2+x3t3+xyzt3
  mstd(j);
↦ [1]:
↦    _[1]=3x2+yz
↦    _[2]=4y3+xzt
↦    _[3]=6z5+xyt3
↦    _[4]=xyzt3
↦    _[5]=y2z2t3
↦    _[6]=yz3t4
↦    _[7]=xz3t4
↦    _[8]=yz2t7
↦    _[9]=xz2t7
↦    _[10]=y2zt7
↦    _[11]=xy2t7
↦ [2]:
↦    _[1]=3x2+yz
↦    _[2]=4y3+xzt
↦    _[3]=6z5+xyt3
↦    _[4]=xyzt3
```

See Section 4.5 [ideal], page 79; Section 5.1.89 [minbase], page 218; Section 4.13 [module], page 111; Section 5.1.147 [std], page 266.

5.1.98 mult

Syntax: `mult` (ideal_expression)
`mult` (module_expression)

Type: int

Purpose: computes the degree of the monomial ideal, resp. module, generated by the leading monomials of the input.
If the input is a standard basis of a homogeneous ideal then it returns the degree of this ideal.
If the input is a standard basis of an ideal in a (local) ring with respect to a local degree ordering then it returns the multiplicity of the ideal (in the sense of Samuel, with respect to the maximal ideal).

Example:

```
  ring r=32003,(x,y),ds;
```

```
  poly f=(x3+y5)^2+x2y7;
  ideal i=std(jacob(f));
  mult(i);
↦ 46
  mult(std(f));
↦ 6
```

See Section 5.1.19 [degree], page 167; Section 5.1.24 [dim], page 170; Section 4.5 [ideal], page 79; Section 5.1.147 [std], page 266; Section 5.1.163 [vdim], page 280.

5.1.99 nameof

Syntax: `nameof` (expression)

Type: string

Purpose: returns the name of an expression as string.

Example:

```
  int i=9;
  string s=nameof(i);
  s;
↦ i
  nameof(s);
↦ s
  nameof(i+1); //returns the empty string:
↦
  nameof(basering);
↦ basering
  basering;
↦    ? `basering` is undefined
↦    ? error occurred in or before ./examples/nameof.sing line 7: `  baseri\
   ng;`
  ring r;
  nameof(basering);
↦ r
```

See Section 5.1.100 [names], page 225; Section 5.1.131 [reservedName], page 250; Section 5.1.156 [typeof], page 277.

5.1.100 names

Syntax: `names` ( )
`names` (ring_name)
`names` (package_name)
`names` (level)

Type: list of strings

Purpose: returns the names of all user-defined variables which are ring independent (this includes the names of procedures) or, in the second case, which belong to the given ring. The third case restricts the variables to the given level.

package_name can be `Current`, `Top` or any other identifier of type package.

Example:

```
  int i=9;
  ring r;
  poly f;
  package p;
  int p::j;
  poly g;
  setring r;
  list l=names();
  l[1..3];
↦ l p r
  names(r);
↦ [1]:
↦    g
↦ [2]:
↦    f
  names(p);
↦ [1]:
↦    j
  names(0);
↦ [1]:
↦    l
↦ [2]:
↦    p
↦ [3]:
↦    r
↦ [4]:
↦    i
↦ [5]:
↦    mathicgb_prOrder
↦ [6]:
↦    mathicgb
↦ [7]:
↦    Singmathic
↦ [8]:
↦    min
↦ [9]:
↦    max
↦ [10]:
↦    datetime
↦ [11]:
↦    weightKB
↦ [12]:
↦    fprintf
↦ [13]:
↦    printf
↦ [14]:
↦    sprintf
↦ [15]:
↦    quotient4
↦ [16]:
↦    quotient5
↦ [17]:
↦    quotient3
```

```
↦ [18]:
↦    quotient2
↦ [19]:
↦    quotient1
↦ [20]:
↦    quot
↦ [21]:
↦    res
↦ [22]:
↦    groebner
↦ [23]:
↦    qslimgb
↦ [24]:
↦    hilbRing
↦ [25]:
↦    par2varRing
↦ [26]:
↦    quotientList
↦ [27]:
↦    stdhilb
↦ [28]:
↦    stdfglm
↦ [29]:
↦    Standard
↦ [30]:
↦    flintZ
↦ [31]:
↦    flint_poly_Q
↦ [32]:
↦    Float
↦ [33]:
↦    crossprod
↦ [34]:
↦    ZZ
↦ [35]:
↦    QQ
↦ [36]:
↦    Top
```

See Section 5.1.99 [nameof], page 225; Section 5.1.131 [reservedName], page 250.

5.1.101 ncols

Syntax: `ncols (` matrix_expression `)`
`ncols (` intmat_expression `)`
`ncols (` ideal_expression `)`

Type: int

Purpose: returns the number of columns of a matrix, an intmat, or the number of given generators of the ideal, including zeros.

Note: `size`(ideal) counts the number of generators which are different from zero. (Use `nrows` to get the number of rows of a given matrix or intmat.)

Example:

```
  ring r;
  matrix m[5][6];
  ncols(m);
↦ 6
  ideal i=x,0,y;
  ncols(i);
↦ 3
  size(i);
↦ 2
```

See Section 4.12 [matrix], page 107; Section 5.1.104 [nrows], page 229; Section 5.1.140 [size], page 260.

5.1.102 npars

Syntax: `npars` `(` ring_name `)`

Type: int

Purpose: returns the number of parameters of a ring.

Example:

```
  ring r=(23,t,v),(x,a(1..7)),lp;
  // the parameters are t,v
  npars(r);
↦ 2
```

See Section 5.1.111 [par], page 235; Section 5.1.113 [parstr], page 236; Section 4.19 [ring], page 125.

5.1.103 nres

Syntax: `nres` `(` ideal_expression`,` int_expression `)`
`nres` `(` module_expression`,` int_expression `)`

Type: resolution

Purpose: computes a free resolution of an ideal or module M which is minimized from the second module on (by the standard basis method).

More precisely, let A_1=matrix(M), then `nres` computes a free resolution of $coker(A_1) = F_0/M$

$$... \longrightarrow F_2 \xrightarrow{A_2} F_1 \xrightarrow{A_1} F_0 \longrightarrow F_0/M \longrightarrow 0,$$

where the columns of the matrix A_1 are the given set of generators of M. If the int expression k is not zero then the computation stops after k steps and returns a list of modules $M_i = \texttt{module}(A_i),\ i = 1, \ldots, k$.
`nres(M,0)` returns a list of n modules where n is the number of variables of the basering. Let `list L=nres(M,0);` then `L[1]=M` is identical to the input, `L[2]` is a minimal set of generators for the first syzygy module of `L[1]`, etc. (`L[i]` $= M_i$ in the notations from above).

Example:

```
  ring r=31991,(t,x,y,z,w),ls;
  ideal M=t2x2+tx2y+x2yz,t2y2+ty2z+y2zw,
          t2z2+tz2w+xz2w,t2w2+txw2+xyw2;
  resolution L=nres(M,0);
  L;
↦  1      4      15      18      7      1
↦ r <--  r <--  r <--   r <--   r <--  r
↦
↦ 0      1      2       3       4      5
↦ resolution not minimized yet
↦
```

See Section 5.1.46 [fres], page 185; Section 5.1.56 [hres], page 195; Section 4.5 [ideal], page 79; Section 5.1.81 [lres], page 212; Section 4.13 [module], page 111; Section 5.1.96 [mres], page 223; [res], page 721; Section 4.18 [resolution], page 124; Section 5.1.145 [sres], page 264.

5.1.104 nrows

Syntax: `nrows` (matrix_expression)
`nrows` (intmat_expression)
`nrows` (intvec_expression)
`nrows` (module_expression)
`nrows` (vector_expression)

Type: int

Purpose: returns the number of rows of a matrix, an intmat or an intvec, resp. the minimal rank of a free module in which the given module or vector lives (the index of the last non-zero component).

Note: Use `ncols` to get the number of columns of a given matrix or intmat.

Example:

```
  ring R;
  matrix M[2][3];
  nrows(M);
↦ 2
  nrows(freemodule(4));
↦ 4
  module m=[0,0,1];
  nrows(m);
↦ 3
  nrows([0,x,0]);
↦ 2
```

See Section 5.1.49 [gen], page 187; Section 4.12 [matrix], page 107; Section 4.13 [module], page 111; Section 5.1.101 [ncols], page 227; Section 4.21 [vector], page 131.

5.1.105 numerator

Syntax: `numerator` (number_expression)

Type: number

Purpose: returns the numerator of a number.

Example:

```
ring r = 0, x, dp;
number n = 3/2;
numerator(n);
↦ 3
```

See Section 5.1.9 [cleardenom], page 161; [content], page 733; Section 5.1.21 [denominator], page 168.

5.1.106 nvars

Syntax: `nvars` (ring_name)

Type: int

Purpose: returns the number of variables of a ring.

Example:

```
ring r=(23,t,v),(x,a(1..7)),ls;
// the variables are x,a(1),...,a(7)
nvars(r);
↦ 8
```

See Section 5.1.102 [npars], page 228; Section 4.19 [ring], page 125; Section 5.1.160 [var], page 279; Section 5.1.162 [varstr], page 280.

5.1.107 open

Syntax: `open` (link_expression)

Type: none

Purpose: opens a link.

Example:

```
link l="ssi:tcp localhost:"+system("Singular");
open(l); // start SINGULAR "server" on localhost in batchmode
close(l); // shut down SINGULAR server
```

See Section 5.1.10 [close], page 161; Section 4.9 [link], page 95.

5.1.108 option

Syntax: `option ()`

Type: string

Purpose: lists all defined options.

Syntax: `option` (option_name)

Type: none

Purpose: sets an option.

Note: To disable an option, use the prefix `no`.

Syntax: `option ( get )`

Type: intvec

Purpose: dumps the state of all options to an intvec.

Syntax: `option ( set,` intvec_expression `)`

Type: none

Purpose: restores the state of all options from an intvec (produced by `option(get)`).

Values: The following options are used to manipulate the behavior of computations and act like boolean switches. Use the prefix `no` to disable an option. Notice that some options are ring dependent and reset to their default values on a change of the current basering.

`none` turns off all options (including the `prompt` option).

`warn` be aware of pitfalls. See Section 3.9.7 [option(warn)], page 71.

`returnSB` the functions `syz`, `intersect` (2 arguments), `quotient` return a standard base instead of a generating set if `returnSB` is set. This option should not be used for `lift`.

`fastHC` tries to find the highest corner of the staircase (HC) as fast as possible during a standard basis computation (only used for local orderings).

`infRedTail`
local normal form computations will not use the ecart to avoid possibly infinite tail reductions: should only be used with extreme care.
By default, it is only set in the case of a zero-dimensional ideal.

`intStrategy`
avoids division of coefficients during standard basis computations. This option is ring dependent. By default, it is set for rings with characteristic 0 and not set for all other rings.

`lazy` uses a more lazy approach in std computations, which was used in SINGULAR version before 2-0 (and which may lead to faster or slower computations, depending on the example)

`length` select shorter reducers in std computations,

`notRegularity`
disables the regularity bound for `res` and `mres` (see Section 5.1.128 [regularity], page 247).

`notSugar` turns off sugar strategy during standard basis computation and reduction.

`notBuckets`
disables the bucket representation of polynomials during standard basis computations. This option usually decreases the memory consumption but increases the computation time. It should only be set for memory-critical standard basis computations.

`prot` shows protocol information indicating the progress during the following computations: `facstd`, `fglm`, `groebner`, `intersect`, `lres`, `mres`, `minres`, `mstd`, `res`, `slimgb`, `sres`, `std`, `stdfglm`, `stdhilb`, `syz`. See below for more details.

`qringNF` simplifies modulo the current `qring` in all assignments.

`redSB` computes a reduced standard basis in any standard basis computation (in rings with global ior local orderings, See Section 5.1.62 [interred], page 199 for the discussion of reduced for local orderings))

`redTail` reduction of the tails of polynomials during standard basis computations. This option is ring dependent. By default, it is set for rings with global degree orderings and not set for all other rings. This option changes the reduction strategy and may decrease/increase time and memory consumption - it does not ensure tail reduction on the result - use redSB for that.

`redThrough`
for inhomogenous input, polynomial reductions during standard basis computations are never postponed, but always finished through. This option is ring dependent. By default, it is set for rings with global degree orderings and not set for all other rings. This option changes the reduction strategy and may decrease/increase time and memory consumption.

`sugarCrit`
uses criteria similar to the homogeneous case to keep more pairs which would be excluded by other criteria but which may be useful for downstream computations. This option changes the strategy for criteria and selection and may decrease/increase time and memory consumption.

`weightM` automatically computes suitable weights for the weighted ecart and the weighted sugar method.

`cancelunit`
avoids to divide polynomials by non-constant units in `std` in the local case. Should usually not be used.

`contentSB`
avoids to divide by the content of a polynomial in `std` and related algorithms. Should usually not be used.

`intersectElim`
prefers elimination to compute intersections (experimental, will be removed in the next release). Should usually not be used.

`intersectSyz`
prefers syzygy methods to compute intersections (experimental, will be removed in the next release). Should usually not be used.

The following options, which also control computations, are special, since they are not manipulated by the `option` command but by a direct assignment of a value. Reset the option by assigning the value 0; the command `option(none)` will not reset them! If there is a non-zero value assigned, the command `option()` prints the option.

`multBound`
a multiplicity bound is set (see Section 5.3.4 [multBound], page 298).

`degBound` a degree bound is set (see Section 5.3.1 [degBound], page 297).

The last set of options controls the output of SINGULAR:

`Imap` shows the mapping of variables with the fetch and imap commands.

`debugLib` warns about syntax errors when loading a library.

`defRes` shows the names of the syzygy modules while converting `resolution` to `list`

`loadLib` shows loading of libraries (set by default).

`loadProc` shows loading of procedures from libraries.

`mem` shows memory usage in square brackets (see Section 5.1.87 [memory], page 217).

`notWarnSB`
do not warn about using a generating set instead of a standard basis.

`prompt` shows prompt (`>`, resp. `.`) if ready for input (default).

`reading` shows the number of characters read from a file.

`redefine` warns about variable redefinitions (set by default).

`usage` shows correct usage in error messages (set by default).

Example:

```
  option(prot);
  option();
↦ //options: prot redefine usage prompt
  option(notSugar);
  option();
↦ //options: prot notSugar redefine usage prompt
  option(noprot);
  option();
↦ //options: notSugar redefine usage prompt
  option(none);
  option();
↦ //options: none
  ring r=0,x,dp;
  degBound=22;
  option();
↦ //options: degBound redTail redThrough intStrategy
  intvec i=option(get);
  option(none);
  option(set,i);
  option();
↦ //options: degBound redTail redThrough intStrategy
```

The output reported on `option(prot)` has the following meaning:

(command)	(character)	(meaning)
`facstd`	`F`	found a new factor
		all other characters: like the output of `std` and `reduce`
`fglm`	`.`	basis monomial found
	`+`	edge monomial found
	`-`	border monomial found

`groebner`		all characters: like the output of `std/slimgb`
`lres`	`.`	minimal syzygy found
	n	slanted degree, i.e., row of Betti matrix
	`(mn)`	calculate in module n
	`g`	pair found giving reductum and syzygy
`mres`	`[d]`	computations of the d-th syzygy module
		all other characters: like the output of `std`
`minres`	`[d]`	minimizing of the d-th syzygy module
`mstd`		all characters: like the output of `std`
`reduce`	`r`	reduced a leading term
	`t`	reduced a non-leading term
`res`	`[d]`	computations of the d-th syzygy module
		all other characters: like the output of `std`
`slimgb`	`M[n,m]`	parallel reduction of n elements with m non-zero output elements
	`.`	postponed a reduction of a pair/S-polynomial
	`b`	exchange of a reductor by a 'better' one
	`e`	a new reductor with non-minimal leading term
	`(n)`	n critical pairs are still to be reduced
	d	the maximal degree of the leading terms is currently d
`sres`	`.`	syzygy found
	`(n)`	n elements remaining
	`[n]`	finished module n
`std`	`[m:n]`	internal ring change to polynomial representation with exponent bound m and n words in exponent vector
	`s`	found a new element of the standard basis
	`-`	reduced a pair/S-polynomial to 0
	`.`	postponed a reduction of a pair/S-polynomial
	`h`	used Hilbert series criterion
	`H(d)`	found a 'highest corner' of degree d, no need to consider higher degrees
	`(n)`	n critical pairs are still to be reduced
	`(S:n)`	doing complete reduction of n elements
	d	the degree of the leading terms is currently d
`stdfglm`		all characters in first part: like the output of `std`
		all characters in second part: like the output of `fglm`
`stdhilb`		all characters: like the output of `std`
`syz`		all characters: like the output of `std`

See Section 5.3.1 [degBound], page 297; Section 5.3.4 [multBound], page 298; Section 5.1.147 [std], page 266.

5.1.109 ord

Syntax: `ord` (poly_expression)
`ord` (vector_expression)

Type: int

Purpose: returns the (weighted) degree of the initial term of a polynomial or a vector; the weights are the weights used for the first block of the ring ordering.

Note: `ord(0)` is `-1`.
In a global degree ordering `ord` is the same as `deg`.

Example:

```
  ring r=7,(x,y),wp(2,3);
  ord(0);
↦ -1
  poly f=x2+y3;  // weight on y is 3
  ord(f),deg(f);
↦ 9 9
  ring R=7,(x,y),ws(2,3);
  poly f=x2+y3;
  ord(f),deg(f);
↦ 4 9
  vector v=[x2,y];
  ord(v),deg(v);
↦ 3 4
```

See Section 5.1.18 [deg], page 167; Section 4.16 [poly], page 118; Section 4.21 [vector], page 131.

5.1.110 ordstr

Syntax: `ordstr` (ring_name)

Type: string

Purpose: returns the description of the monomial ordering of the ring.

Example:

```
  ring r=7,(x,y),wp(2,3);
  ordstr(r);
↦ wp(2,3),C
```

See Section 5.1.7 [charstr], page 160; Section 5.1.113 [parstr], page 236; Section 4.19 [ring], page 125; Section 5.1.162 [varstr], page 280.

5.1.111 par

Syntax: `par` (int_expression)

Type: number

Purpose: `par(n);` returns the n-th parameter of the basering.

Example:

```
  ring r=(0,a,b,c),(x,y,z),dp;
  char(r);  // char to get the characteristic
↦ 0
  par(2);   // par to get the n-th parameter
↦ (b)
```

See Section 5.1.5 [char], page 159; Section 5.1.102 [npars], page 228; Section 5.1.113 [parstr], page 236; Section 4.19 [ring], page 125; Section 5.1.160 [var], page 279.

5.1.112 pardeg

Syntax: `pardeg` (number_expression)

Type: int

Purpose: returns the degree of a number considered as a polynomial in the ring parameters.

Example:

```
  ring r=(0,a,b,c),(x,y,z),dp;
  pardeg(a^2*b);
↦ 3
```

See Section 5.1.18 [deg], page 167; Section 4.14 [number], page 114; Section 4.19 [ring], page 125; Section 5.1.160 [var], page 279.

5.1.113 parstr

Syntax: `parstr` (ring_name)
`parstr` (int_expression)
`parstr` (ring_name, int_expression)

Type: string

Purpose: returns the list of parameters of the ring as a string or the name of the n-th parameter where n is given by the int_expression.
If the ring_name is omitted, the basering is used, thus `parstr(n)` is equivalent to `parstr(basering,n)`.

Example:

```
  ring r=(7,a,b,c),(x,y),wp(2,3);
  parstr(r);
↦ a,b,c
  parstr(2);
↦ b
  parstr(r,3);
↦ c
```

See Section 5.1.7 [charstr], page 160; Section 5.1.102 [npars], page 228; Section 5.1.110 [ordstr], page 235; Section 4.19 [ring], page 125; Section 5.1.162 [varstr], page 280.

5.1.114 preimage

Syntax: `preimage` (map)
`preimage` (ring_name, map_name, ideal_name)
`preimage` (ring_name, ideal_expression, ideal_name)

Type: ring
ideal

Purpose: returns the source ring of a map (in the first case) or returns the preimage of an ideal under a given map.
The second argument has to be a map from the basering to the given ring (or an ideal defining such a map), and the ideal has to be an ideal in the given ring.

Note: As `preimage` is handling ideals (not polynomials), the result of a preimage calculation of a principal ideal is (the closure of) the preimage of the ideal, not that of the polynomial.

Example:

```
  ring r1=32003,(x,y,z,w),lp;
  ring r=32003,(x,y,z),dp;
  ideal i=x,y,z;
  ideal i1=x,y;
  ideal i0=0;
  map f=r1,i;
  nameof (preimage (f));
↦ r1
  setring r1;
  ideal i1=preimage(r,f,i1);
  i1;
↦ i1[1]=w
↦ i1[2]=y
↦ i1[3]=x
  // the kernel of f
  preimage(r,f,i0);
↦ _[1]=w
  // or, use:
  kernel(r,f);
↦ _[1]=w
```

See Section 4.5 [ideal], page 79; Section 5.1.68 [kernel], page 204; Section 4.11 [map], page 104; Section 4.19 [ring], page 125.

5.1.115 prime

Syntax: `prime` (int_expression)

Type: int

Purpose: returns the largest prime less than or equal to the argument; returns 2 for all arguments smaller than 3.

Example:

```
  prime(320000);
↦ 319993
  prime(32004);
↦ 32003
  prime(0);
↦ 2
  prime(-1);
↦ 2
```

See Section D.2.3 [general_lib], page 726; Section 4.6 [int], page 83.

5.1.116 primefactors

Syntax: `primefactors` (int/bigint/number_expression)
`primefactors` (int/bigint/number_expression , int_expression)

Type: list

Purpose: returns the prime factorisation up to an optionally given bound, b, on the prime factors
When called with int(s)/bigint(s), no ring needs to be active.
When called with numbers these are assumed to be integers in a polynomial ring over Q.
The method finds all prime factors of an integer n. n' will contain the sign, be zero, or the rest (when a bound is given) respectively. The returned list contains the following information: The returned list contains the following information:
L[1][i] = i-th prime factor (in ascending order),
L[2][i] = multiplicity of L[1][i],
L[3] = n'

Example:

```
  bigint n = bigint(7)^12 * bigint(37)^6 * 121;
  primefactors(n);
↦ [1]:
↦    [1]:
↦       7
↦    [2]:
↦       11
↦    [3]:
↦       37
↦ [2]:
↦    [1]:
↦       12
↦    [2]:
↦       2
↦    [3]:
↦       6
↦ [3]:
↦    1
  primefactors(n,25);
↦ [1]:
↦    [1]:
↦       7
↦    [2]:
↦       11
↦ [2]:
↦    [1]:
↦       12
↦    [2]:
↦       2
↦ [3]:
↦    2565726409
```

See Section 5.1.115 [prime], page 237.

5.1.117 print

Syntax: `print` (expression)
`print` (expression, `"betti"`)
`print` (expression, format_string)

Type: string

Purpose: The first form prints the expression.
The second form prints the graded Betti numbers from a matrix. The Betti numbers are printed in a matrix-like format where the entry d in row i and column j is the minimal number of generators in degree $i+j$ of the j-th syzygy module of R^n/M (the 0th and 1st syzygy module of R^n/M is R^n and M, resp.).
The last form returns the printed output as a string depending on the format string iwhich determines the format to use to generate the string.

The following format strings are supported:

`"%s"` returns `string`(expression),

`"%2s"` similar to `"%s"`, except that newlines are inserted after every comma and at the end,

`"%l"` similar to `"%s"`, except that each object is embraced by its type such that it can be directly used for "cutting and pasting",

`"%2l"` similar to `"%l"`, except that newlines are inserted after every comma and at the end,

`"%;"` returns the string equivalent to typing `expression;`

`"%t"` returns the string equivalent to typing `type expression;`

`"%p"` returns the string equivalent to typing `print(expression);`

`"%b"` returns the string equivalent to typing `print(expression, "betti");`

`"betti"` is not a format string.

Example:

```
  ring r=0,(x,y,z),dp;
  module m=[1,y],[0,x+z];
  m;
↦ m[1]=y*gen(2)+gen(1)
↦ m[2]=x*gen(2)+z*gen(2)
  print(m);  // the columns generate m
↦ 1,0,
↦ y,x+z
  string s=print(m,"%s"); s;
↦ y*gen(2)+gen(1),x*gen(2)+z*gen(2)
  s=print(m,"%2s"); s;
↦ y*gen(2)+gen(1),
↦ x*gen(2)+z*gen(2)
↦
  s=print(m,"%l"); s;
↦ module(y*gen(2)+gen(1),x*gen(2)+z*gen(2))
  s=print(m,"%;"); s;
↦ m[1]=y*gen(2)+gen(1)
↦ m[2]=x*gen(2)+z*gen(2)
↦
  s=print(m,"%t"); s;
```

```
↦ // m module , rk 2
↦ m[1]=y*gen(2)+gen(1)
↦ m[2]=x*gen(2)+z*gen(2)
  s=print(m,"%p"); s;
↦ 1,0,
↦ y,x+z
  intmat M=betti(mres(m,0));
  print(M,"betti");
↦            0     1
↦ ------------------
↦     0:     1     1
↦ ------------------
↦ total:     1     1
↦
  list l=r,M;
  s=print(l,"%s"); s;
↦ (QQ),(x,y,z),(dp(3),C),1,1
  s=print(l,"%2s"); s;
↦ (QQ),(x,y,z),(dp(3),C),
↦ 1,1
↦
  s=print(l,"%l"); s;
↦ list("(QQ),(x,y,z),(dp(3),C)",intmat(intvec(1,1 ),1,2))
```

See Section 3.5.5 [Type conversion and casting], page 45; Section 5.1.4 [betti], page 157; Section 5.1.16 [dbprint], page 165; [fprintf], page 721; [printf], page 721; Section 5.3.7 [short], page 299; [sprintf], page 721; Section 4.20.3 [string type cast], page 129; Section 5.1.155 [type], page 277.

5.1.118 printf

Procedure from library `standard.lib` (see Section D.1 [standard_lib], page 721).

Syntax: `printf (` string_expression `[,` any_expressions`] )`

Return: none

Purpose: `printf(fmt,...);` performs output formatting. The first argument is a format control string. Additional arguments may be required, depending on the content of the control string. A series of output characters is generated as directed by the control string; these characters are displayed (i.e., printed to standard out).
The control string `fmt` is simply text to be copied, except that the string may contain conversion specifications.
Type `help print;` for a listing of valid conversion specifications. As an addition to the conversions of `print`, the `%n` and `%2` conversion specification does not consume an additional argument, but simply generates a newline character.

Note: If one of the additional arguments is a list, then it should be enclosed once more into a `list()` command, since passing a list as an argument flattens the list by one level.

Example:

```
  ring r=0,(x,y,z),dp;
module m=[1,y],[0,x+z];
intmat M=betti(mres(m,0));
list l=r,m,matrix(M);
printf("s:%s,l:%l",1,2);
```

```
↦ s:1,l:int(2)
printf("s:%s",l);
↦ s:(QQ),(x,y,z),(dp(3),C)
printf("s:%s",list(l));
↦ s:(QQ),(x,y,z),(dp(3),C),y*gen(2)+gen(1),x*gen(2)+z*gen(2),1,1
printf("2l:%2l",list(l));
↦ 2l:list("(QQ),(x,y,z),(dp(3),C)",
↦ module(y*gen(2)+gen(1),
↦ x*gen(2)+z*gen(2)),
↦ matrix(ideal(1,
↦ 1),1,2))
↦
printf("%p",matrix(M));
↦ 1,1
printf("%;",matrix(M));
↦ _[1,1]=1
↦ _[1,2]=1
↦
printf("%b",M);
↦            0     1
↦ ------------------
↦     0:     1     1
↦ ------------------
↦ total:     1     1
↦
```

See also: [fprintf], page 721; Section 5.1.117 [print], page 238; [sprintf], page 721; Section 4.20 [string], page 128.

5.1.119 prune

Syntax: `prune` (module_expression)

Type: module

Purpose: returns the module minimally embedded in a free module such that the corresponding factor modules are isomorphic.

Note: If for the input module the attribute `"isHomog"` is set, `prune` also sets the attribute `"isHomog"`.
For non-global orderings, only reduction steps with constant units are performed. Hence, the returned module does not need to be minimal.

Example:

```
  ring r=0,(x,y,z),dp;
  module m=gen(1),gen(3),[x,y,0,z],[x+y,0,0,0,1];
  print(m);
↦ 1,0,x,x+y,
↦ 0,0,y,0,
↦ 0,1,0,0,
↦ 0,0,z,0,
↦ 0,0,0,1
  print(prune(m));
↦ y,
↦ z
```

See Section 4.13 [module], page 111.

5.1.120 qhweight

Syntax: `qhweight` (ideal_expression)

Type: intvec

Purpose: computes the weight vector of the variables for a quasihomogeneous ideal. If the input is not weighted homogeneous, an intvec of zeros is returned.

Example:

```
ring h1=32003,(t,x,y,z),dp;
ideal i=x4+y3+z2;
qhweight(i);
↦ 0,3,4,6
```

See Section 4.5 [ideal], page 79; Section 4.8 [intvec], page 92; Section 5.1.167 [weight], page 283.

5.1.121 qrds

Syntax: `qrds (` matrix_expression, number_expression, number_expression, number_expression)

Type: list

Purpose: computes all eigenvalues with multiplicities of the given matrix by performing the numeric QR double shift algorithm involving Hessenberg form and householder transformations.
This method expects the ground field to be the complex numbers, and all matrix entries to be real numbers, i.e., elements of this ground field with the imaginary part equal to zero.
If the algorithm works, then it returns a list with two entries which are again lists of the same size:
_[1][i] is the i-th mutually distinct eigenvalue that was found,
_[2][i] is the (int) multiplicity of _[1][i].
If the algorithm does not work (due to an ill-posed matrix), a list with the single entry (int)0 is returned.
The first number argument is used for detection of deflation in the actual QR double shift algorithm. The second number argument is used for ending Heron's iteration whenever square roots are being computed. And the third number argument is used to distinguish between distinct eigenvalues: When the Euclidean distance between two computed eigenvalues is less then this number, then they will be regarded equal, resulting in a higher multiplicity of the corresponding eigenvalue. (A good choice for all three number arguments is a small value like e.g. 10^(-100).)

Example:

```
ring r=(complex,50),(dummy),dp;
matrix A[3][3]=-10,37,-5,-14,51,-10,-29,99,-18;
bigint b = bigint(10)^100; number t = 1/b;
list L=qrds(A,t,t,t); L;
↦ [1]:
↦    [1]:
↦ (3+i*2)
↦    [2]:
```

```
↦ (3-i*2)
↦    [3]:
↦ 17
↦ [2]:
↦    [1]:
↦       1
↦    [2]:
↦       1
↦    [3]:
↦       1
```

5.1.122 quote

Syntax: `quote` (expression)

Type: none

Purpose: prevents expressions from evaluation. Used only in connections with write to ssi links, prevents evaluation of an expression before sending it to an other Singular process. Within a quoted expression, the quote can be "undone" by an `eval` (i.e., each eval "undoes" the effect of exactly one quote).

Example:

```
  link l="ssi:w example.ssi";
  ring r=0,(x,y,z),ds;
  ideal i=maxideal(3);
  ideal j=x7,x2,z;
  option(prot);
  // compute i+j before writing, but not std
  write (l, quote(std(eval(i+j))));
  close(l);
  // now read it in again and evaluate:
  read(l);
↦ [65535:1]1(12)s2(11)s3(10)--s(7)s(6)-----7-
↦ product criterion:4 chain criterion:0
↦ _[1]=z
↦ _[2]=x2
↦ _[3]=xy2
↦ _[4]=y3
  close(l);
```

See Section 4.9.5 [Ssi links], page 97; Section 5.1.28 [eval], page 174; Section 5.1.169 [write], page 283.

5.1.123 quotient

Syntax: `quotient` (ideal_expression, ideal_expression)
`quotient` (module_expression, module_expression)

Type: ideal

Syntax: `quotient` (module_expression, ideal_expression)

Type: module

Purpose: computes the ideal quotient, resp. module quotient. Let `R` be the basering, `I`,`J` ideals and `M` a module in R^n. Then `quotient(I,J)`$= \{a \in R \mid aJ \subset I\}$,

`quotient(M,J)`$= \{b \in R^n \mid bJ \subset M\}$.

Example:

```
ring r=181,(x,y,z),(c,ls);
ideal id1=maxideal(3);
ideal id2=x2+xyz,y2-z3y,z3+y5xz;
ideal id6=quotient(id1,id2);
id6;
↦ id6[1]=z
↦ id6[2]=y
↦ id6[3]=x
quotient(id2,id1);
↦ _[1]=z2
↦ _[2]=yz
↦ _[3]=y2
↦ _[4]=xz
↦ _[5]=xy
↦ _[6]=x2
module m=x*freemodule(3),y*freemodule(2);
ideal id3=x,y;
quotient(m,id3);
↦ _[1]=[1]
↦ _[2]=[0,1]
↦ _[3]=[0,0,x]
```

See Section 5.1.39 [fglmquot], page 181; Section 4.5 [ideal], page 79; Section 4.13 [module], page 111.

5.1.124 random

Syntax: `random` (int_expression, int_expression)

Type: int

Purpose: returns a random integer between the integer given by the first int_expression and the one given by the second int_expression.

Syntax: `random` (int_expression, int_expression, int_expression)

Type: intmat

Purpose: returns a random intmat where the size is given by the second (number of rows) and third argument (number of columns). The absolute value of the entries of the matrix is smaller than or equal to the integer given as the first argument.

Note: The random generator can be set to a startvalue with the function `system`, resp. by a command line option. The current value of the random generator is `system("random")`.
Internally a random generator with values in 1 to 2^31 - 2 and a full period is used, max-min may not be larger than 2^31-2.

Example:

```
  random(1,1000);
↦ 35
  random(1,2,3);
```

```
↦ 0,0,0,
↦ 1,1,-1
  system("random",210);  // start random generator with 210
  random(-1000,1000);
↦ 707
  random(-1000,1000);
↦ 284
  system("random",210);
  random(-1000,1000);    // the same random values again
↦ 707
```

See Section 3.1.6 [Command line options], page 19; Section 4.6 [int], page 83; Section 4.7 [intmat], page 89; Section 5.1.151 [system], page 270.

5.1.125 rank

Syntax: `rank` (matrix_expression) #*`rank` (matrix_expression `,1)`

Type: int

Purpose: returns the rank of a given matrix which is filled with elements of the ground field. The first variant uses a LU-decomposition, the second a row-echelon form.

Note: The function works by computing the row echelon form of the matrix using the same algorithm as for `ludecomp`.

Example:

```
  ring s = 0, x, dp;
  matrix A[100][100];
  int i; int j; int r;
  for (i = 1; i <= 100; i++)
  {
    for (j = 1; j <= 100; j++)
    {
      A[i, j] = random(-10, 10);
    }
  }
  r = rank(A); r;
↦ 100
```

See Section 5.1.82 [ludecomp], page 213.

5.1.126 read

Syntax: `read` (link_expression)
for DBM links:
`read` (link_expression)
`read` (link_expression`,` string_expression)

Type: any

Purpose: reads data from a link.
For ASCII links, the content of the entire file is returned as a string. If the ASCII link is the empty string, `read` reads from standard input.
For ssi links, one expression is read from the link and returned after evaluation. See Section 4.9.5 [Ssi links], page 97.

For ssi links the `read` command blocks as long as there is no data to be read from the link. The `status` command can be used to check whether or not there is data to be read.
For DBM links, a `read` with one argument returns the value of the next entry in the data base, and a `read` with two arguments returns the value to the key given as the second argument from the data base. See Section 4.9.7 [DBM links], page 100.

Example:

```
  ring r=32003,(x,y,z),dp;
  ideal i=x+y,z3+22y;
  // write the ideal i to the file save_i
  write(":w save_i",i);
  ring r0=0,(x,y,z),Dp;
  // create an ideal k equal to the content
  // of the file save_i
  string s="ideal k="+read("save_i")+";";
  execute(s);
  k;
↦ k[1]=x+y
↦ k[2]=z3+22y
```

See Section 5.1.31 [execute], page 175; Section 5.1.50 [getdump], page 188; Section 4.9 [link], page 95; Section 5.1.146 [status], page 265; Section 5.1.169 [write], page 283.

5.1.127 reduce

Syntax:
`reduce` (poly_expression, ideal_expression)
`reduce` (poly_expression, ideal_expression, int_expression)
`reduce` (poly_expression, poly_expression, ideal_expression)
`reduce` (vector_expression, ideal_expression)
`reduce` (vector_expression, ideal_expression, int_expression)
`reduce` (vector_expression, module_expression)
`reduce` (vector_expression, module_expression, int_expression)
`reduce` (vector_expression, poly_expression, module_expression)
`reduce` (ideal_expression, ideal_expression)
`reduce` (ideal_expression, ideal_expression, int_expression)
`reduce` (ideal_expression, matrix_expression, ideal_expression)
`reduce` (module_expression, ideal_expression)
`reduce` (module_expression, ideal_expression, int_expression)
`reduce` (module_expression, module_expression)
`reduce` (module_expression, module_expression, int_expression)
`reduce` (module_expression, matrix_expression, module_expression)
`reduce` (poly/vector/ideal/module, ideal/module, int, intvec)
`reduce` (ideal, matrix, ideal, int)
`reduce` (poly, poly, ideal, int)
`reduce` (poly, poly, ideal, int, intvec)

Type: the type of the first argument

Purpose: reduces a polynomial, vector, ideal or module to its normal form with respect to an ideal or module represented by a standard basis. Returns 0 if and only if the polynomial (resp. vector, ideal, module) is an element (resp. subideal, submodule) of the ideal (resp. module). The result may have no meaning if the second argument is not a standard

basis.
The third (optional) argument of type int modifies the behavior:

0 default

1 consider only the leading term and do no tail reduction.

2 tail reduction:n the local/mixed ordering case: reduce also with bad ecart

4 reduce without division, return possibly a non-zero constant multiple of the remainder

If a second argument `u` of type poly or matrix is given, the first argument `p` is replaced by `p/u`. This works only for zero dimensional ideals (resp. modules) in the third argument and gives, even in a local ring, a reduced normal form which is the projection to the quotient by the ideal (resp. module). One may give a degree bound in the fourth argument with respect to a weight vector in the fifth argument in order have a finite computation. If some of the weights are zero, the procedure may not terminate!

Note: The commands `reduce` and `NF` are synonymous.

Example:

```
  ring r1 = 0,(z,y,x),ds;
  poly s1=2x5y+7x2y4+3x2yz3;
  poly s2=1x2y2z2+3z8;
  poly s3=4xy5+2x2y2z3+11x10;
  ideal i=s1,s2,s3;
  ideal j=std(i);
  reduce(3z3yx2+7y4x2+yx5+z12y2x2,j);
↦ -yx5+2401/81y14x2+2744/81y11x5+392/27y8x8+224/81y5x11+16/81y2x14
  reduce(3z3yx2+7y4x2+yx5+z12y2x2,j,1);
↦ -yx5+z12y2x2
  // 4 arguments:
  ring rs=0,x,ds;
  // normalform of 1/(1+x) w.r.t. (x3) up to degree 5
  reduce(poly(1),1+x,ideal(x3),5);
↦ // ** _ is no standard basis
↦ 1-x+x2
```

See Section 5.1.25 [division], page 171; Section 4.5 [ideal], page 79; Section 4.13 [module], page 111; Section 4.16.3 [poly operations], page 120; Section 5.1.147 [std], page 266; Section 4.21 [vector], page 131.

5.1.128 regularity

Syntax: `regularity` (list_expression)
`regularity` (resolution_expression)

Type: int

Purpose: computes the regularity of a homogeneous ideal, resp. module, from a minimal resolution given by the argument.
Let $0 \to \bigoplus_a K[x]e_{a,n} \to \ldots \to \bigoplus_a K[x]e_{a,0} \to I \to 0$ be a minimal resolution of I considered with homogeneous maps of degree 0. The regularity is the smallest number s with the property $\deg(e_{a,i}) \leq s+i$ for all i.

Note: If applied to a non minimal resolution only an upper bound is returned.
If the input to the commands `res` and `mres` is homogeneous the regularity is computed and used as a degree bound during the computation unless `option(notRegularity);` is given.

Example:

```
  ring rh3=32003,(w,x,y,z),(dp,C);
  poly f=x11+y10+z9+x5y2+x2y2z3+xy3*(y2+x)^2;
  ideal j=homog(jacob(f),w);
  def jr=res(j,0);
  regularity(jr);
↦ 25
  // example for upper bound behaviour:
  list jj=jr;
  regularity(jj);
↦ 25
  jj=nres(j,0);
  regularity(jj);
↦ 27
  jj=minres(jj);
  regularity(jj);
↦ 25
```

See Section 5.1.46 [fres], page 185; Section 4.10 [list], page 102; Section 5.1.91 [minres], page 220; Section 5.1.96 [mres], page 223; Section 5.1.108 [option], page 230; [res], page 721; Section 4.18 [resolution], page 124; Section 5.1.145 [sres], page 264.

5.1.129 repart

Syntax: `repart (` number_expression `)`

Type: number

Purpose: returns the real part of a number from a complex ground field,
returns its argument otherwise.

Example:

```
  ring r=(complex,i),x,dp;
  repart(1+2*i);
↦ 1
```

See Section 5.1.58 [impart], page 196.

5.1.130 res

Procedure from library `standard.lib` (see Section D.1 [standard_lib], page 721).

Syntax: `res (` ideal_expression`,` int_expression `[,` any_expression `])`
`res (` module_expression`,` int_expression `[,` any_expression `])`

Type: resolution

Purpose: computes a (possibly minimal) free resolution of an ideal or module using a heuristically chosen method.
The second (int) argument (say `k`) specifies the length of the resolution. If it is not positive then `k` is assumed to be the number of variables of the basering.
If a third argument is given, the returned resolution is minimized.

Depending on the input, the returned resolution is computed using the following methods:

quotient rings:
: `nres` (classical method using syzygies) , see Section 5.1.103 [nres], page 228.

homogeneous ideals and k=0:
: `lres` (La'Scala's method), see Section 5.1.81 [lres], page 212.

not minimized resolution and (homogeneous input with k not 0, or local rings):
: `sres` (Schreyer's method), see Section 5.1.145 [sres], page 264.

all other inputs:
: `mres` (classical method), see Section 5.1.96 [mres], page 223.

Note: Accessing single elements of a resolution may require some partial computations to be finished and may therefore take some time.

See also Section 5.1.4 [betti], page 157; Section 5.1.46 [fres], page 185; Section 5.1.56 [hres], page 195; Section 4.5 [ideal], page 79; Section 5.1.81 [lres], page 212; Section 5.1.91 [minres], page 220; Section 4.13 [module], page 111; Section 5.1.96 [mres], page 223; Section 5.1.103 [nres], page 228; Section 4.18 [resolution], page 124; Section 5.1.145 [sres], page 264.

Example:

```
  ring r=0,(x,y,z),dp;
ideal i=xz,yz,x3-y3;
def l=res(i,0); // homogeneous ideal: uses lres
l;
↦  1      3      2
↦ r <--  r <--  r
↦
↦ 0      1      2
↦
print(betti(l), "betti"); // input to betti may be of type resolution
↦            0     1     2
↦ ------------------------
↦     0:     1     -     -
↦     1:     -     2     1
↦     2:     -     1     1
↦ ------------------------
↦ total:     1     3     2
↦
l[2];         // element access may take some time
↦ _[1]=-x*gen(1)+y*gen(2)
↦ _[2]=-x2*gen(2)+y2*gen(1)+z*gen(3)
i=i,x+1;
l=res(i,0);   // inhomogeneous ideal: uses mres
l;
↦  1      3      3      1
↦ r <--  r <--  r <--  r
↦
↦ 0      1      2      3
↦ resolution not minimized yet
↦
ring rs=0,(x,y,z),ds;
ideal i=imap(r,i);
```

```
def l=res(i,0); // local ring not minimized: uses sres
l;
↦   1       1
↦ rs <--  rs
↦
↦ 0       1
↦ resolution not minimized yet
↦
res(i,0,0);     // local ring and minimized: uses mres
↦   1       1
↦ rs <--  rs
↦
↦ 0       1
↦
```

5.1.131 reservedName

Syntax: `reservedName ()`

Type: none

Syntax: `reservedName (` string_expression `)`

Type: int

Purpose: prints a list of all reserved identifiers (first form) or tests whether the string is a reserved identifier (second form). This includes blackbox/newstruct types.

Example:

```
reservedName();
↦ ... // output skipped
  reservedName("ring");
↦ 1
  reservedName("xyz");
↦ 0
```

See Section 5.1.100 [names], page 225; Section 4.20 [string], page 128.

5.1.132 resultant

Syntax: `resultant (` poly_expression`,` poly_expression`,` ring_variable `)`

Type: poly

Purpose: computes the resultant of the first and second argument with respect to the variable given as the third argument.

Example:

```
  ring r=32003,(x,y,z),dp;
  poly f=3*(x+2)^3+y;
  poly g=x+y+z;
  resultant(f,g,x);
↦ 3y3+9y2z+9yz2+3z3-18y2-36yz-18z2+35y+36z-24
```

See Section 4.16 [poly], page 118; Section 4.19 [ring], page 125.

5.1.133 ringlist

Syntax: `ringlist` (ring_expression)

Type: list

Purpose: decomposes a ring/qring into a list of 4 (or 6 in the non-commutative case, see Section 7.3.24 [ringlist (plural)], page 352) components. It is identical to `ring_list` with the exception of the first list entry.

1. the field description in the following format:

   for Q, Z/p: the characteristic, type int (0 or prime number)

   for real, complex: a list of:
   the characteristic, type int (always 0)
   the precision, type list (2 integers: external, internal precision)
   the name of the imaginary unit, type string

   for transcendental or algebraic extensions: described as a ringlist (that is, as list L with 4 entries: L[1] the characteristic, L[2] the names of the parameters, L[3] the monomial ordering for the ring of parameters (default: lp), L[4] the minimal polynomial (as ideal))

   for Z, Z/n, Z/n^m a list ["integer", [n, m]] with:
   the base n is of type int or bigint (if not given n = 0, Z/0 = Z)
   the exponent m is of type int (if not given m = 1)

2. the names of the variables (a list L of strings: L[i] is the name of the i-th variable)
3. the monomial ordering (a list L of lists): each block L[i] consists of

   the name of the ordering (string)

   parameters specifying the ordering and the size of the block (intvec : typically the weights for the variables [default: 1])
4. the quotient ideal.

From a list of such structure, a new ring may be defined by the command `ring` (see the following example). If the attribute "maxExp" of the ring is different from the default 32767, it is also set for the list.

Note: `All data which depends on a ring belong to the current ring,`
not to a ring which can be contructed fron a modified list. These data will be mapped via fetch to the ring to be constructed.

Example:

```
  ring r = 0,(x(1..3)),dp;
  list l = ringlist(r);
  l;
↦ [1]:
↦    0
↦ [2]:
↦    [1]:
↦       x(1)
↦    [2]:
↦       x(2)
↦    [3]:
↦       x(3)
↦ [3]:
```

```
↦    [1]:
↦       [1]:
↦          dp
↦       [2]:
↦          1,1,1
↦    [2]:
↦       [1]:
↦          C
↦       [2]:
↦          0
↦ [4]:
↦    _[1]=0
 //  Now change l and create a new ring, by
 //- changing the base field to the function field with parameter a,
 //- introducing one extra variable y,
 //- defining the block ordering (dp(2),wp(3,4)).
 //- define the minpoly after creating the function field
 l[1]=list(0,list("a"),list(list("lp",1)),ideal(0));
 l[2][size(l[2])+1]="y";
 l[3][3]=l[3][2]; // save the module ordering
 l[3][1]=list("dp",intvec(1,1));
 l[3][2]=list("wp",intvec(3,4));
 attrib(l,"maxExp",100); // and lower the limit for exponents to 100
 def ra = ring(l);     //creates the newring
 ra; setring ra;
↦ // coefficients: QQ(a)
↦ // number of vars : 4
↦ //        block   1 : ordering dp
↦ //                  : names    x(1) x(2)
↦ //        block   2 : ordering wp
↦ //                  : names    x(3) y
↦ //                  : weights  3 4
↦ //        block   3 : ordering C
 attrib(ra,"maxExp");
↦ 32767
 list lra = ringlist(ra);
 lra[1][4]=ideal(a2+1);
 def Ra = ring(lra);
 setring Ra; Ra;
↦ // coefficients: QQ[a]/(a^2+1)
↦ // number of vars : 4
↦ //        block   1 : ordering dp
↦ //                  : names    x(1) x(2)
↦ //        block   2 : ordering wp
↦ //                  : names    x(3) y
↦ //                  : weights  3 4
↦ //        block   3 : ordering C
```

See Section 4.19.1 [qring], page 125; Section 4.19 [ring], page 125; Section 5.1.134 [ring_list], page 252.

5.1.134 ring_list

Syntax: `ring_list` (ring_expression)

Type: list

Purpose: decomposes a ring/qring into a list of 4 (or 6 in the non-commutative case, see Section 7.3.24 [ringlist (plural)], page 352) components. It is identical to `ringlist` with the exception of the first list entry.

1. the field description as `cring`
2. the names of the variables (a list L of strings: L[i] is the name of the i-th variable)
3. the monomial ordering (a list L of lists): each block L[i] consists of

   the name of the ordering (string)

   parameters specifying the ordering and the size of the block (intvec : typically the weights for the variables [default: 1])
4. the quotient ideal.

From a list of such structure, a new ring may be defined by the command `ring` (see the following example).

Note: `All data which depends on a ring belong to the current ring,` not to a ring which can be contructed fron a modified list. These data will be mapped via fetch to the ring to be constructed.

Example:

```
ring r = 0,(x(1..3)),dp;
list l = ring_list(r);
l;
↦ [1]:
↦ QQ
↦ [2]:
↦    [1]:
↦       x(1)
↦    [2]:
↦       x(2)
↦    [3]:
↦       x(3)
↦ [3]:
↦    [1]:
↦       [1]:
↦          dp
↦       [2]:
↦          1,1,1
↦    [2]:
↦       [1]:
↦          C
↦       [2]:
↦          0
↦ [4]:
↦    _[1]=0
//  Now change l and create a new ring, by
//- changing the base field to ZZ/32003
//- introducing one extra variable y,
//- defining the block ordering (dp(2),wp(3,4)).
//- define the minpoly after creating the function field
l[1]=ZZ/32003;
l[2][size(l[2])+1]="y";
```

```
  l[3][3]=l[3][2]; // save the module ordering
  l[3][1]=list("dp",intvec(1,1));
  l[3][2]=list("wp",intvec(3,4));
  def ra = ring(l);     //creates the newring
  ra; setring ra;
↦ // coefficients: ZZ/32003
↦ // number of vars : 4
↦ //        block   1 : ordering dp
↦ //                  : names    x(1) x(2)
↦ //        block   2 : ordering wp
↦ //                  : names    x(3) y
↦ //                  : weights  3 4
↦ //        block   3 : ordering C
```

See Section 4.19.1 [qring], page 125; Section 4.19 [ring], page 125; Section 5.1.133 [ringlist], page 251.

5.1.135 rvar

Syntax: `rvar` (name)
`rvar` (poly_expression)
`rvar` (string_expression)

Type: int

Purpose: returns the number of the variable if the name/polynomial is a ring variable of the basering or if the string is the name of a ring variable of the basering; returns 0 if not. Hence the return value of `rvar` can also be used in a boolean context to check whether the variable exists.

Example:

```
  ring r=29,(x,y,z),lp;
  rvar(x);
↦ 1
  rvar(r);
↦ 0
  rvar(y);
↦ 2
  rvar(var(3));
↦ 3
  rvar("x");
↦ 1
```

See Section 5.1.17 [defined], page 166; Section 4.19 [ring], page 125; Section 5.1.160 [var], page 279; Section 5.1.162 [varstr], page 280.

5.1.136 sba

Syntax: `sba` (ideal_expression)
`sba` (ideal_expression, int_expression, int_expression)

Type: ideal

Purpose: returns a standard basis of an ideal with respect to the monomial ordering of the basering. A standard basis is a set of generators such that the leading terms generate the leading ideal, resp. module.

Use optional second and third arguments of type `int` to determine the respective variant of the signature-based standard basis algorithm:
The second argument specifies the internal module order `sba` uses:

`0`: induced Schreyer order on the signatures, non-incremental computation of the basis

`1`: position over term order, incremental computation of the basis

`2`: term over position order, non-incremental computation

`3`: Schreyer-weighted degree over index over leading term

The third argument specifies the rewrite order `sba` uses:

`0`: using the rewrite order described in `http://dx.doi.org/10.1016/j.jsc.2010.06.019`

`1`: using the rewrite order described in `http://dx.doi.org/10.1016/j.jsc.2011.05.004`

The standard call of `sba(i)` corresponds to `sba(i,0,1)`.

Note: The standard basis is computed with an optimized version of known signature-based algorithms like Faugere's F5 Algorithm. Whereas the correctness of the algorithms is only guaranteed for global orderings, timings for pure lexicographical orderings can be slow. In this situation you should try to compute the basis w.r.t. the graded reverse-lexicographic ordering and then convert to a basis for the lexicographical ordering using other methods (see Section 5.1.38 [fglm], page 180 and see Section D.4.7 [grwalk_lib], page 745). If the algorithms tend to use too much memory, you should try the other implemented standard basis algorithms (see Section 5.1.147 [std], page 266, see [groebner], page 721, and see Section 5.1.141 [slimgb], page 261).
Note that the behaviour of `sba` on an example can be rather different depending on which variant you choose (second and third argument).

Example:

```
// incremental F5 computation
ring r=32003,(x,y,z),dp;
poly s1=1x2y+151xyz10+169y21;
poly s2=1xz14+6x2y4+3z24;
poly s3=5y10z10x+2y20z10+y10z20+11x3;
ideal i=s1,s2,s3;
ideal j=sba(i,1,0);
// non-incremental F5 computation
ring rhom=32003,(x,y,z,h),dp;
ideal i=homog(imap(r,i),h);
ideal j=sba(i,0,0);
// non-incremental signature-based computation
ring whom=32003,(x,y,z),dp;
ideal i=fetch(r,i);
ideal j=sba(i);
```

See Section 5.1.38 [fglm], page 180; [groebner], page 721; Section 4.5 [ideal], page 79; Section 4.19 [ring], page 125; Section 5.1.141 [slimgb], page 261; Section 5.1.147 [std], page 266.

5.1.137 setring

Syntax: `setring` ring_name

Type: none

Purpose: changes the basering to another (already defined) ring.

Example:

```
ring r1=0,(x,y),lp;
// the basering is r1
ring r2=32003,(a(1..8)),ds;
// the basering is r2
setring r1;
// the basering is again r1
nameof(basering);
↦ r1
listvar();
↦ // r2                         [0]  ring
↦ // r1                         [0]  *ring
```

Use in procedures:
All changes of the basering by a definition of a new ring or a `setring` command in a procedure are local to this procedure. Use `keepring` to move a ring, which is local to a procedure, up by one nesting level.

See Section 5.2.11 [keepring], page 293; Section 4.19.1 [qring], page 125; Section 4.19 [ring], page 125.

5.1.138 simplex

Syntax: `simplex (` matrix_expression`,` int_expression`,` int_expression`,` int_expression`,` int_expression`,` int_expression`)`

Type: list

Purpose: perform the simplex algorithm for the tableau given by the input, e.g. `simplex (`M, m, n, m1, m2, m3 `)`:

M matrix of numbers :
first row describing the objective function (maximize problem), the remaining rows describing constraints;

m, n, m1, m2, m3 int :
n = number of variables; m = total number of constraints; m1 = number of inequalities "<=" (rows 2 ... m1+1 of M); m2 = number of inequalities ">=" (rows m1+2 ... m1+m2+1 of M); m3 = number of equalities.

The following assumptions are made:

* ground field is of type `(real,N)`, N>=4;

* the matrix M is of size m x n;

* m=m1+m2+m3;

* the entries M[2,1] ,..., M[m+1,1] are non-negative;

* the variables x(i) are non-negative;

* a row b, a(1) ,..., a(n) corresponds to b+a(1)x(1)+...+a(n)x(n);

* for a <=, >=, or == constraint: add "in mind" >=0, <=0, or ==0.

The output is a list L with

* L[1] = matrix

* L[2] = int:
  0 = finite solution found; 1 = unbounded; -1 = no solution; -2 = error occured;
* L[3] = intvec :
  L[3][k] = number of variable which corresponds to row k+1 of L[1];
* L[4] = intvec :
  L[4][j] = number of variable which is represented by column j+1 of L[1] ("non-basis variable");
* L[5] = int :
  number of constraints (= m);
* L[6] = int :
  number of variables (= n).

The solution can be read off the first column of L[1] as it is done by the procedure [simplexOut], page 791 in `solve.lib`.

Example:

```
ring r = (real,10),(x),lp;

// consider the max. problem:
//
//    maximize  x(1) + x(2) + 3*x(3) - 0.5*x(4)
//
//  with constraints:   x(1) +          2*x(3)          <= 740
//                               2*x(2)          - 7*x(4) <=   0
//                                 x(2) -   x(3) + 2*x(4) >=   0.5
//                      x(1) +   x(2) +   x(3) +   x(4)  =   9
//
matrix sm[5][5]=(  0, 1, 1, 3,-0.5,
                 740,-1, 0,-2, 0,
                   0, 0,-2, 0, 7,
                 0.5, 0,-1, 1,-2,
                   9,-1,-1,-1,-1);

int n = 4;  // number of constraints
int m = 4;  // number of variables
int m1= 2;  // number of <= constraints
int m2= 1;  // number of >= constraints
int m3= 1;  // number of == constraints
simplex(sm, n, m, m1, m2, m3);
↦ [1]:
↦    _[1,1]=17.025
↦    _[1,2]=-0.95
↦    _[1,3]=-0.05
↦    _[1,4]=1.95
↦    _[1,5]=-1.05
↦    _[2,1]=730.55
↦    _[2,2]=0.1
↦    _[2,3]=-0.1
↦    _[2,4]=-1.1
↦    _[2,5]=0.9
```

```
↦     _[3,1]=3.325
↦     _[3,2]=-0.35
↦     _[3,3]=-0.15
↦     _[3,4]=0.35
↦     _[3,5]=0.35
↦     _[4,1]=0.95
↦     _[4,2]=-0.1
↦     _[4,3]=0.1
↦     _[4,4]=0.1
↦     _[4,5]=0.1
↦     _[5,1]=4.725
↦     _[5,2]=-0.55
↦     _[5,3]=0.05
↦     _[5,4]=0.55
↦     _[5,5]=-0.45
↦  [2]:
↦     0
↦  [3]:
↦     5,2,4,3
↦  [4]:
↦     1,6,8,7
↦  [5]:
↦     4
↦  [6]:
↦     4
```

See [simplexOut], page 791.

5.1.139 simplify

Syntax: `simplify` (poly_expression, int_expression)
`simplify` (vector_expression, int_expression)
`simplify` (ideal_expression, int_expression)
`simplify` (module_expression, int_expression)

Type: the type of the first argument

Purpose: returns the "simplified" first argument depending on the simplification rules specified by the second argument. The simplification rules are the following functions:

`1` normalize (divide by leading coefficient if this is a unit of the ground field/ring).

`2` erase zero generators/columns.

`4` erase copies of earlier listed generators/columns.

`8` erase generators/columns which a skalar multiples (w.r.t. ground field/ring) of earlier listed generators/columns.

`16` erase generators/columns whose leading monomials are copies of leading monomials of earlier listed generators/columns such that the coefficients of both leading terms are units in the ground field/ring.

`32` erase generators/columns whose leading terms are divisible by leading terms of other (not necessarily earlier) listed generators/columns.

64 normalize each coefficient of every monomial (of every polynomial)

Example:

```
ring r=0,(x,y,z),(c,dp);
ideal i=0,2x,2x,4x,3x+y,5x2;
simplify(i,1);
↦ _[1]=0
↦ _[2]=x
↦ _[3]=x
↦ _[4]=x
↦ _[5]=x+1/3y
↦ _[6]=x2
simplify(i,2);
↦ _[1]=2x
↦ _[2]=2x
↦ _[3]=4x
↦ _[4]=3x+y
↦ _[5]=5x2
simplify(i,4);
↦ _[1]=0
↦ _[2]=2x
↦ _[3]=0
↦ _[4]=4x
↦ _[5]=3x+y
↦ _[6]=5x2
simplify(i,8);
↦ _[1]=0
↦ _[2]=2x
↦ _[3]=0
↦ _[4]=0
↦ _[5]=3x+y
↦ _[6]=5x2
simplify(i,16);
↦ _[1]=0
↦ _[2]=2x
↦ _[3]=0
↦ _[4]=0
↦ _[5]=0
↦ _[6]=5x2
simplify(i,32);
↦ _[1]=0
↦ _[2]=2x
↦ _[3]=0
↦ _[4]=0
↦ _[5]=0
↦ _[6]=0
simplify(i,32+2+1);
↦ _[1]=x
matrix A[2][3]=x,0,2x,y,0,2y;
simplify(A,2+8); // by automatic conversion to module
↦ _[1]=[x,y]
```

See Section 4.5 [ideal], page 79; Section 4.13 [module], page 111; Section 4.16 [poly], page 118; Section 4.21 [vector], page 131.

5.1.140 size

Syntax: `size` (string_expression)
`size` (bigint_expression)
`size` (number_expression)
`size` (intvec_expression)
`size` (intmat_expression)
`size` (poly_expression)
`size` (vector_expression)
`size` (ideal_expression)
`size` (module_expression)
`size` (matrix_expression)
`size` (list_expression)
`size` (resolution_expression)
`size` (ring_expression)

Type: int

Purpose: depends on the type of argument:

ideal or module
: returns the number of (non-zero) generators.

string, intvec, list or resolution
: returns the length, i.e., the number of characters, entries or elements.

poly or vector
: returns the number of monomials.

matrix or intmat
: returns the number of entries (rows*columns).

ring
: returns the number of elements in the ground field (for Z/p and algebraic extensions) or -1

number or bigint
: returns 0 for 0 or the number of words

Example:

```
  string s="hello";
  size(s);
↦ 5
  intvec iv=1,2;
  size(iv);
↦ 2
  ring r=0,(x,y,z),lp;
  poly f=x+y+z;
  size(f);
↦ 3
  vector v=[x+y,0,0,1];
  size(v);
↦ 3
  ideal i=f,y;
  size(i);
↦ 2
  module m=v,[0,1],[0,0,1],2*v;
```

```
  size(m);
↦ 4
  matrix mm[2][2];
  size(mm);
↦ 4
  ring r1=(2,a),x,dp;
  minpoly=a4+a+1;
  size(r1);
↦ 16
```

See Section 4.5 [ideal], page 79; Section 4.7 [intmat], page 89; Section 4.8 [intvec], page 92; Section 4.13 [module], page 111; Section 5.1.101 [ncols], page 227; Section 5.1.104 [nrows], page 229; Section 4.16 [poly], page 118; Section 4.20 [string], page 128; Section 4.21 [vector], page 131.

5.1.141 slimgb

Syntax:
`slimgb` (ideal_expression)
`slimgb` (module_expression)

Type: ideal or module

Purpose: Section A.2.3 [slim Groebner bases], page 642

Returns a Groebner basis of an ideal or module with respect to the monomial ordering of the basering (which has to be global).

Note: The algorithm is designed to keep polynomials slim (short with small coefficients). For details see `http://www.mathematik.uni-kl.de/~zca/Reports_on_ca/35/paper_35_full.ps.gz`. A reduced Groebner basis is returned if option(redSB) is set (see [option(redSB)], page 231). To view the progress of long running computations, use option(prot) (see [option(prot)], page 231).

Example:

```
  ring r=2,(x,y,z),lp;
  poly s1=z*(x*y+1);
  poly s2=x2+x;
  poly s3=y2+y;
  ideal i=s1,s2,s3;
  slimgb(i);
↦ _[1]=y2+y
↦ _[2]=x2+x
↦ _[3]=yz+z
↦ _[4]=xz+z
```

See [groebner], page 721; Section 4.5 [ideal], page 79; Section 5.1.108 [option], page 230; Section 4.19 [ring], page 125; Section 5.1.147 [std], page 266.

5.1.142 sortvec

Syntax: `sortvec` (ideal_expression)
`sortvec` (module_expression)

Type: intvec

Purpose: computes the permutation `v` which orders the ideal, resp. module, `I` by its initial terms, starting with the smallest, that is, `I(v[i]) < I(v[i+1])` for all `i`.

Example:

```
ring r=0,(x,y,z),dp;
ideal I=y,z,x,x3,xz;
sortvec(I);
↦ 2,1,3,5,4
```

See Section D.2.3 [general_lib], page 726.

5.1.143 sqrfree

Syntax: `sqrfree` (poly_expression)
`sqrfree` (poly_expression, `0`)
`sqrfree` (poly_expression, `2`)

Type: list of ideal and intvec

Syntax: `sqrfree` (poly_expression, `1`)

Type: ideal

Syntax: `sqrfree` (poly_expression, `3`)

Type: poly

Purpose: computes the squarefree factors (as an ideal) of the polynomial together with or without the multiplicities (as an intvec) depending on the second argument:

```
0: returns factors and multiplicities, first factor is a constant.
   May also be written with only one argument.
1: returns non-constant factors (no multiplicities).
2: returns non-constant factors and multiplicities.
3: returns the product of non-constant factors, i.e. squarefree part
```

Note: Not implemented for the coefficient fields real and finite fields of type `(p^n,a)`.

Example:

```
ring r=3,(x,y,z),dp;
poly f=(x-y)^3*(x+z)*(y-z);
sqrfree(f);
↦ [1]:
↦    _[1]=1
↦    _[2]=-xy+xz-yz+z2
↦    _[3]=-x+y
↦ [2]:
↦    1,1,3
sqrfree(f,1);
↦ _[1]=-xy+xz-yz+z2
↦ _[2]=-x+y
sqrfree(f,2);
↦ [1]:
↦    _[1]=-xy+xz-yz+z2
↦    _[2]=-x+y
↦ [2]:
↦    1,3
sqrfree(f,3);
↦ x2y-xy2-x2z-xyz-y2z-xz2+yz2
```

See Section 5.1.35 [factorize], page 178.

5.1.144 sprintf

Procedure from library `standard.lib` (see Section D.1 [standard_lib], page 721).

Syntax: `sprintf (` string_expression `[,` any_expressions `] )`

Return: string

Purpose: `sprintf(fmt,...);` performs output formatting. The first argument is a format control string. Additional arguments may be required, depending on the content of the control string. A series of output characters is generated as directed by the control string; these characters are returned as a string.
The control string `fmt` is simply text to be copied, except that the string may contain conversion specifications.
Type `help print;` for a listing of valid conversion specifications. As an addition to the conversions of `print`, the `%n` and `%2` conversion specification does not consume an additional argument, but simply generates a newline character.

Note: If one of the additional arguments is a list, then it should be wrapped in an additional `list()` command, since passing a list as an argument flattens the list by one level.

Example:

```
  ring r=0,(x,y,z),dp;
module m=[1,y],[0,x+z];
intmat M=betti(mres(m,0));
list l = r, m, M;
string s = sprintf("s:%s,%n l:%l", 1, 2); s;
↦ s:1,
↦  l:int(2)
s = sprintf("s:%n%s", l); s;
↦ s:
↦ (QQ),(x,y,z),(dp(3),C)
s = sprintf("s:%2%s", list(l)); s;
↦ s:
↦ (QQ),(x,y,z),(dp(3),C),y*gen(2)+gen(1),x*gen(2)+z*gen(2),1,1
s = sprintf("2l:%n%2l", list(l)); s;
↦ 2l:
↦ list("(QQ),(x,y,z),(dp(3),C)",
↦ module(y*gen(2)+gen(1),
↦ x*gen(2)+z*gen(2)),
↦ intmat(intvec(1,1 ),1,2))
↦
s = sprintf("%p", list(l)); s;
↦ [1]:
↦    // coefficients: QQ
↦ // number of vars : 3
↦ //        block   1 : ordering dp
↦ //                  : names    x y z
↦ //        block   2 : ordering C
↦ [2]:
↦    _[1]=y*gen(2)+gen(1)
↦    _[2]=x*gen(2)+z*gen(2)
↦ [3]:
```

```
↦    1,1
s = sprintf("%;", list(l)); s;
↦ [1]:
↦    // coefficients: QQ
↦ // number of vars : 3
↦ //        block   1 : ordering dp
↦ //                  : names    x y z
↦ //        block   2 : ordering C
↦ [2]:
↦    _[1]=y*gen(2)+gen(1)
↦    _[2]=x*gen(2)+z*gen(2)
↦ [3]:
↦    1,1
↦ 
s = sprintf("%b", M); s;
↦            0     1
↦ ------------------
↦     0:     1     1
↦ ------------------
↦ total:     1     1
↦ 
```

See also: [fprintf], page 721; Section 5.1.117 [print], page 238; [printf], page 721; Section 4.20 [string], page 128.

5.1.145 sres

Syntax: `sres (` ideal_expression`,` int_expression `)`
`sres (` module_expression`,` int_expression `)`

Type: resolution

Purpose: computes a free resolution of an ideal or module with Schreyer's method. The ideal, resp. module, has to be a standard basis. More precisely, let M be given by a standard basis and $A_1 =$ `matrix`(M). Then `sres` computes a free resolution of $coker(A_1) = F_0/M$

$$... \longrightarrow F_2 \xrightarrow{A_2} F_1 \xrightarrow{A_1} F_0 \longrightarrow F_0/M \longrightarrow 0.$$

If the int expression k is not zero then the computation stops after k steps and returns a list of modules (given by standard bases) $M_i =$ `module`(A_i), i=1..k.
`sres(M,0)` returns a list of n modules where n is the number of variables of the basering.

Even if `sres` does not compute a minimal resolution, the `betti` command gives the true betti numbers! In many cases of interest `sres` is much faster than any other known method. Let `list L=sres(M,0);` then `L[1]=M` is identical to the input, `L[2]` is a standard basis with respect to the Schreyer ordering of the first syzygy module of `L[1]`, etc. (L[i] $= M_i$ in the notations from above.)

Note: Accessing single elements of a resolution may require some partial computations to be finished and may therefore take some time.

Example:

```
ring r=31991,(t,x,y,z,w),ls;
ideal M=t2x2+tx2y+x2yz,t2y2+ty2z+y2zw,
        t2z2+tz2w+xz2w,t2w2+txw2+xyw2;
M=std(M);
```

```
  resolution L=sres(M,0);
  L;
↦  1      35      141      209      141      43      4
↦ r <--  r <--   r <--    r <--    r <--    r <--   r
↦
↦ 0      1       2        3        4        5       6
↦ resolution not minimized yet
↦
  print(betti(L),"betti");
↦            0     1     2     3     4     5
↦ ------------------------------------------
↦     0:     1     -     -     -     -     -
↦     1:     -     -     -     -     -     -
↦     2:     -     -     -     -     -     -
↦     3:     -     4     -     -     -     -
↦     4:     -     -     -     -     -     -
↦     5:     -     -     -     -     -     -
↦     6:     -     -     6     -     -     -
↦     7:     -     -     9    16     2     -
↦     8:     -     -     -     2     5     1
↦ ------------------------------------------
↦ total:     1     4    15    18     7     1
↦
```

See Section 5.1.4 [betti], page 157; Section 5.1.46 [fres], page 185; Section 5.1.56 [hres], page 195; Section 4.5 [ideal], page 79; Section 4.6 [int], page 83; Section 5.1.81 [lres], page 212; Section 5.1.91 [minres], page 220; Section 4.13 [module], page 111; Section 5.1.96 [mres], page 223; [res], page 721; Section 5.1.152 [syz], page 275.

5.1.146 status

Syntax: `status` (link_expression`,` string_expression)

Type: string

Syntax: `status` (link_expression`,` string_expression`,` string_expression)

Type: int

Purpose: returns the status of the link as asked for by the second argument. If a third argument is given, the result of the comparison to the status string is returned: `(status(l,s1)==s2)` is equivalent to `status(l,s1,s2)`.
The following string expressions are allowed:

`"name"` the name string given by the definition of the link (usually the filename)

`"type"` returns `"ASCII"`, `"DBM"` or `"ssi"`

`"open"` returns `"yes"` or `"no"`

`"openread"`
returns `"yes"` or `"no"`

`"openwrite"`
returns `"yes"` or `"no"`

`"read"` returns `"ready"` or `"not ready"` `"write"` returns `"ready"` or `"not ready"`

`"mode"` returns (depending on the type of the link and its status) `""`,`"w"`,`"a"`,`"r"` or `"rw"`

`"exists"` returns `"yes"` or `"no"`: existence of the filename for ASCII/ssi links

Syntax: `status` (list_expression, int_expression)

Type: int

Purpose: the list should be a list L of links, the second argument a timeout in 1/10 seconds. Returns

`-2` select returns an error

`-1` all links are closed/at eof

`0` timeout

`>0` (at least) L[i] is ready

Example:

```
  link l=":w example.txt";
  status(l,"write");
↦ not ready
  open(l);
  status(l,"write","ready");
↦ 1
  close(l);
```

See Section 4.9 [link], page 95; Section 5.1.107 [open], page 230; Section 5.1.126 [read], page 245; Section 5.1.169 [write], page 283.

5.1.147 std

Syntax: `std` (ideal_expression)
`std` (module_expression)
`std` (ideal_expression, intvec_expression)
`std` (module_expression, intvec_expression)
`std` (ideal_expression, intvec_expression, intvec_expression)
`std` (module_expression, intvec_expression, intvec_expression)
`std` (ideal_expression, poly_expression)
`std` (module_expression, vector_expression)
`std` (ideal_expression, ideal_expression)
`std` (module_expression, module_expression)
`std` (ideal_expression, poly_expression, intvec_expression, intvec_expression)
`std` (module_expression, poly_expression, intvec_expression, intvec_expression)

Type: ideal or module

Purpose: returns a standard basis of an ideal or module with respect to the monomial ordering of the basering. A standard basis is a set of generators such that the leading terms generate the leading ideal, resp. module.
Use an optional second argument of type intvec as Hilbert series (result of `hilb(i,1)`, see Section 5.1.54 [hilb], page 193), if the ideal, resp. module, is homogeneous (Hilbert driven standard basis computation, [stdhilb], page 721). If the ideal is quasihomogeneous with some weights w and if the Hilbert series is computed w.r.t. to these weights,

then use w as third argument.
Use an optional second argument of type poly/vector/ideal, resp. module, to construct the standard basis from an already computed one (given as the first argument) and additional generator(s) (the second argument).
4 arguments `G,p,hv,w` are the combination of the above: standard basis `G`, additional generator `p`, hilbert function `hv` w.r.t. weights `w`.

Note: The standard basis is computed with a (more or less) straight-forward implementation of the classical Buchberger (resp. Mora) algorithm. For global orderings, use the `groebner` command instead (see [groebner], page 721), which heuristically chooses the "best" algorithm to compute a Groebner basis.
To view the progress of long running computations, use `option(prot)` (see [option(prot)], page 231).

Example:

```
  // local computation
  ring r=32003,(x,y,z),ds;
  poly s1=1x2y+151xyz10+169y21;
  poly s2=1xz14+6x2y4+3z24;
  poly s3=5y10z10x+2y20z10+y10z20+11x3;
  ideal i=s1,s2,s3;
  ideal j=std(i);
  degree(j);
↦ // dimension (local)   = 0
↦ // multiplicity = 1512
  // Hilbert driven elimination (standard)
  ring rhom=32003,(x,y,z,h),dp;
  ideal i=homog(imap(r,i),h);
  ideal j=std(i);
  intvec iv=hilb(j,1);
  ring rlex=32003,(x,y,z,h),lp;
  ideal i=fetch(rhom,i);
  ideal j=std(i,iv);
  j=subst(j,h,1);
  j[1];
↦ z64
  // Hilbert driven elimination (ideal is quasihomogeneous)
  intvec w=10,1,1;
  ring whom=32003,(x,y,z),wp(w);
  ideal i=fetch(r,i);
  ideal j=std(i);
  intvec iw=hilb(j,1,w);
  ring wlex=32003,(x,y,z),lp;
  ideal i=fetch(whom,i);
  ideal j=std(i,iw,w);
  j[1];
↦ z64
```

See Section 5.1.33 [facstd], page 176; Section 5.1.38 [fglm], page 180; [groebner], page 721; Section 4.5 [ideal], page 79; Section 5.1.97 [mstd], page 223; Section 5.1.108 [option], page 230; Section 4.19 [ring], page 125; [stdfglm], page 721; [stdhilb], page 721.

5.1.148 stdfglm

Procedure from library `standard.lib` (see Section D.1 [standard_lib], page 721).

Syntax: `stdfglm` (ideal_expression)
`stdfglm` (ideal_expression, string_expression)

Type: ideal

Purpose: computes the standard basis of the ideal in the basering via `fglm` from the ordering given as the second argument to the ordering of the basering. If no second argument is given, "dp" is used. The standard basis for the given ordering (resp. for "dp") is computed via the command groebner except if a further argument "std" or "slimgb" is given in which case std resp. slimgb is used.

Example:

```
   ring r  = 0,(x,y,z),lp;
ideal i = y3+x2,x2y+x2,x3-x2,z4-x2-y;
stdfglm(i);                   //uses fglm from "dp" (with groebner) to "lp"
↦ _[1]=z12
↦ _[2]=yz4-z8
↦ _[3]=y2+y-z8-z4
↦ _[4]=xy-xz4-y+z4
↦ _[5]=x2+y-z4
stdfglm(i,"std");             //uses fglm from "dp" (with std) to "lp"
↦ _[1]=z12
↦ _[2]=yz4-z8
↦ _[3]=y2+y-z8-z4
↦ _[4]=xy-xz4-y+z4
↦ _[5]=x2+y-z4
ring s  = (0,x),(y,z,u,v),lp;
minpoly = x2+1;
ideal i = u5-v4,zv-u2,zu3-v3,z2u-v2,z3-uv,yv-zu,yu-z2,yz-v,y2-u,u-xy2;
weight(i);
↦ 2,3,4,5
stdfglm(i,"(a(2,3,4,5),dp)"); //uses fglm from "(a(2,3,4,5),dp)" to "lp"
↦ _[1]=v2
↦ _[2]=u
↦ _[3]=zv
↦ _[4]=z2
↦ _[5]=yv
↦ _[6]=yz-v
↦ _[7]=y2
```

See also: Section 5.1.38 [fglm], page 180; [groebner], page 721; Section 5.1.141 [slimgb], page 261; Section 5.1.147 [std], page 266; [stdhilb], page 721.

5.1.149 stdhilb

Procedure from library `standard.lib` (see Section D.1 [standard_lib], page 721).

Syntax: `stdhilb` (ideal_expression)
`stdhilb` (module_expression)
`stdhilb` (ideal_expression, intvec_expression)
`stdhilb` (module_expression, intvec_expression)

`stdhilb` (ideal_expression, list of string_expressions, and intvec_expression)

Type: type of the first argument

Purpose: Compute a Groebner basis of the ideal/module in the basering by using the Hilbert driven Groebner basis algorithm. If an argument of type string, stating `"std"` resp. `"slimgb"`, is given, the standard basis computation uses `std` or `slimgb`, otherwise a heuristically chosen method (default)
If an optional second argument w of type intvec is given, w is used as variable weights. If w is not given, it is computed as w[i] = deg(var(i)). If the ideal is homogeneous w.r.t. w then the Hilbert series is computed w.r.t. to these weights.

Theory: If the ideal is not homogeneous compute first a Groebner basis of the homogenization [w.r.t. the weights w] of the ideal/module, then the Hilbert function and, finally, a Groebner basis in the original ring by using the computed Hilbert function. If the given w does not coincide with the variable weights of the basering, the result may not be a groebner basis in the original ring.

Note: 'Homogeneous' means weighted homogeneous with respect to the weights w[i] of the variables var(i) of the basering. Parameters are not converted to variables.

Example:

```
   ring  r = 0,(x,y,z),lp;
ideal i = y3+x2,x2y+x2z2,x3-z9,z4-y2-xz;
ideal j = stdhilb(i); j;
↦ j[1]=z10
↦ j[2]=yz9
↦ j[3]=2y2z4-z8
↦ j[4]=2y3z3-2y2z5-yz7
↦ j[5]=y4+y3z2
↦ j[6]=xz+y2-z4
↦ j[7]=xy2-xz4-y3z
↦ j[8]=x2+y3
ring  r1 = 0,(x,y,z),wp(3,2,1);
ideal  i = y3+x2,x2y+x2z2,x3-z9,z4-y2-xz;  //ideal is homogeneous
ideal j = stdhilb(i,"std"); j;
↦ j[1]=y2+xz-z4
↦ j[2]=x2-xyz+yz4
↦ j[3]=2xz5-z8
↦ j[4]=2xyz4-yz7+z9
↦ j[5]=z10
↦ j[6]=2yz9+z11
//this is equivalent to:
intvec v = hilb(std(i),1);
ideal j1 = std(i,v,intvec(3,2,1)); j1;
↦ j1[1]=y2+xz-z4
↦ j1[2]=x2-xyz+yz4
↦ j1[3]=2xz5-z8
↦ j1[4]=2xyz4-yz7+z9
↦ j1[5]=z10
↦ j1[6]=yz9
size(NF(j,j1))+size(NF(j1,j));            //j and j1 define the same ideal
↦ 0
```

See also: [groebner], page 721; Section 5.1.141 [slimgb], page 261; Section 5.1.147 [std], page 266; [stdfglm], page 721.

5.1.150 subst

Syntax: `subst` (poly_expression`,` variable`,` poly_expression)
`subst` (poly_expression`,` variable`,` poly_expression `,...` variable`,` poly_expression)
`subst` (vector_expression`,` variable`,` poly_expression)
`subst` (ideal_expression`,` variable`,` poly_expression)
`subst` (module_expression`,` variable`,` poly_expression)

Type: poly, vector, ideal or module (corresponding to the first argument)

Purpose: substitutes one or more ring variable(s)/parameter variable(s) by (a) polynomial(s). Note that in the case of more than one substitution pair, the substitutions will be performed sequentially and not simultaneously. The below examples illustrate this behaviour.
Note, that the coefficients must be polynomial when substituting a parameter.

Example:

```
  ring r=0,(x,y,z),dp;
  poly f=x2+y2+z2+x+y+z;
  subst(f,x,y,y,z);   // first substitute x by y, then y by z
↦ 3z2+3z
  subst(f,y,z,x,y);   // first substitute y by z, then x by y
↦ y2+2z2+y+2z
```

See Section 4.5 [ideal], page 79; Section 4.11 [map], page 104; Section 4.13 [module], page 111; Section 4.16 [poly], page 118; [substitute], page 734; Section 4.21 [vector], page 131.

5.1.151 system

Syntax: `system` (string_expression)
`system` (string_expression`,` expression)

Type: depends on the desired function, may be none

Purpose: interface to internal data and the operating system. The string_expression determines the command to execute. Some commands require an additional argument (second form) where the type of the argument depends on the command. See below for a list of all possible commands.

Note: Not all functions work on every platform.

Functions:

`system("alarm"`, int)
: abort the Singular process after computing for that many seconds (system+user cpu time).

`system("absFact"`, poly)
: absolute factorization of the polynomial (from a polynomial ring over a transzedental extension) Returns a list of the ideal of the factors, intvec of multiplicities, ideal of minimal polynomials and the bumber of factors.

`system("blackbox")`
: list all blackbox data types.

`system("browsers");`
: returns a string about available help browsers. See Section 3.1.3 [The online help system], page 15.

`system("bracket",` poly, poly `)`
: returns the Lie bracket [p,q].

`system("btest",` poly, i2 `)`
: internal for shift algebra (with i2 variables): last block of the poly

`system("complexNearZero",` number_expression `)`
: checks for a small value for floating point numbers

`system("contributors")`
: returns names of people who contributed to the SINGULAR kernel as string.

`system("cpu")`
: returns the number of cpus as int (for creating multiple threads/processes). (see `system("--cpus")`).

`system("denom_list")`
: returns the list of denominators (number) which occured in the latest std computationi(s). Is reset to the empty list at ring changes or by this system call.

`system("eigenvals",` matrix `)`
: returns the list of the eigenvalues of the matrix (as ideal, intvec). (see `system("hessenberg")`).

`system("env",` ring `)`
: returns the enveloping algebra (i.e. R tensor R^opp) See `system("opp")`.

`system("executable",` string `)`
: returns the path of the command given as argument or the empty string (for: not found) See `system("Singular")`. See `system("getenv","PATH")`.

`system("freegb",` ideal, i2, i3 `)`
: returns the standrda basis in the shift algebra i(with i3 variables) up to degree i2. See `system("opp")`.

`system("getenv",` string_expression`)`
: returns the value of the shell environment variable given as the second argument. The return type is string.

`system("getPrecDigits")`
: returns the precision for floating point numbers

`system("gmsnf",` ideal, ideal, matrix,int, int `)`
: Gauss-Manin system: for gmspoly.lib, gmssing.lib

`system("HC")`
: returns the degree of the "highest corner" from the last std computation (or 0).

`system("hessenberg",` matrix `)`
: returns the Hessenberg matrix (via QR algorithm).

`system("install",` s1, s2, p3, i4 `)`
: install a new method p3 for s2 for the newstruct type s1. s2 must be a reserved operator with i4 operands (i4 may be 1,2,3; use 4 for more than 3 or a varying number of arguments) See See Section 4.22.4 [Commands for user defined types], page 136.

`system("LLL",` B `)`
: B must be a matrix or an intmat. Interface to NTLs LLL (Exact Arithmetic Variant over ZZ). Returns the same type as the input.
B is an m x n matrix, viewed as m rows of n-vectors. m may be less than, equal to, or greater than n, and the rows need not be linearly independent. B is transformed into an LLL-reduced basis. The first m-rank(B) rows of B are zero.
More specifically, elementary row transformations are performed on B so that the non-zero rows of new-B form an LLL-reduced basis for the lattice spanned by the rows of old-B.

`system("nblocks")` or `system("nblocks",` ring_name `)`
: returns the number of blocks of the given ring, or of the current basering, if no second argument is given. The return type is int.

`system("nc_hilb",` ideal, int, [,...] `)`
: internal support for ncHilb.lib, return nothing

`system("neworder",` ideal `)`
: string of the ring variables in an heurically good order for `char_series`

`system("newstruct")`
: list all newstruct data types.

`system("opp",` ring `)`
: returns the opposite ring.

`system("oppose",` ring R, poly p `)`
: returns the opposite polynomial of p from R.

`system("pcvLAddL",` list, list `)`
: `system("pcvPMulL",` poly, list `)`
`system("pcvMinDeg",` poly `)`
`system("pcvP2CV",` list, int, int `)`
`system("pcvCV2P",` list, int, int `)`
`system("pcvDim",` int, int `)`
`system("pcvBasis",` int, int `)` internal for mondromy.lib

`system("pid")`
: returns the process number as int (for creating unique names).

`system("random")` or `system("random",` int `)`
: returns or sets the seed of the random generator.

`system("reduce_bound",` poly, ideal, int `)`
: or `system("reduce_bound",` ideal, ideal, int `)`
or `system("reduce_bound",` vector, module, int `)`
or `system("reduce_bound",` module, module, int `)` returns the normal-form of the first argument wrt. the second up to the given degree bound (wrt. total degree)

`system("reserve",` int `)`
: reserve a port and listen with the given backlog. (see `system("reservedLink")`).

`system("reservedLink")`
: accept a connect at the reserved port and return a (write-only) link to it. (see `system("reserve")`).

`system("semaphore",` string, int `)`
: operations for semaphores: string may be `"init"`, `"exists"`, `"acquire"`, `"try_acquire"`, `"release"`, `"get_value"`, and int is the number of the semaphore. Returns -2 for wrong command, -1 for error or the result of the command.

`system("semic",` list, list `)`
: or `system("semic",` list, list, int `)` computes from list of spectrum numbers and list of spectrum numbers the semicontinuity index (qh, if 3rd argument is 1).

`system("setenv",`string_expression, string_expression`)`
: sets the shell environment variable given as the second argument to the value given as the third argument. Returns the third argument. Might not be available on all platforms.

`system("sh"`, string_expression `)`
: shell escape, returns the return code of the shell as int. The string is sent literally to the shell.

`system("shrinktest",` poly, i2 `)`
: internal for shift algebra (with i2 variables): shrink the poly

`system("Singular")`
: returns the absolute (path) name of the running Singular as string.

`system("SingularLib")`
: returns the colon seperated library search path name as string.

`system("spadd",` list, list `)`
: or `system("spadd",` list, list, int `)` computes from list of spectrum numbers and list of spectrum numbers the sum of the lists.

`system("spectrum",` poly `)`
: or `system("spectrum",` poly, int `)`

`system("spmul",` list, int `)`
: or `system("spmul",` list, list, int `)` computes from list of spectrum numbers the multiple of it.

`system("std_syz",` module, int `)`
: compute a partial groebner base of a module, stopp after the given column

`system("stest",` poly, i2, i3, i4 `)`
: internal for shift algebra (with i4 variables): shift the poly by i2, up to degree i3

`system("tensorModuleMult",` int, module `)`
: internal for sheafcoh.lib (see id_TensorModuleMult)

`system("twostd",` ideal `)`
: returns the two-sided standard basis of the two-sided ideal.

`system("uname")`
: returns a string identifying the architecture for which Singular was compiled.

`system("version")`
: returns the version number of Singular as int. (Version a-b-c-d returns a*10000+b*1000+c*100+d)

`system("with")`
: without an argument: returns a string describing the current version of Singular, its build options, the used path names and other configurations with a string argument: test for that feature and return an int.

`system("--cpus")`
: returns the number of available cpu cores as int (for using multiple cores). (see `system("cpu")`).

`system("–")`
: prints the values of all options.

`system("–long_option_name")`
: returns the value of the (command-line) option long_option_name. The type of the returned value is either string or int. See Section 3.1.6 [Command line options], page 19, for more info.

`system("–long_option_name", expression)`
: sets the value of the (command-line) option long_option_name to the value given by the expression. Type of the expression must be string, or int. See Section 3.1.6 [Command line options], page 19, for more info. Among others, this can be used for setting the seed of the random number generator, the used help browser, the minimal display time, or the timer resolution.

Example:

```
// a listing of the current directory:
system("sh","ls");
// execute a shell, return to SINGULAR with exit:
system("sh","sh");
string unique_name="/tmp/xx"+string(system("pid"));
unique_name;
↦ /tmp/xx4711
system("uname")
↦ ix86-Linux
system("getenv","PATH");
↦ /bin:/usr/bin:/usr/local/bin
system("Singular");
↦ /usr/local/bin/Singular
// report value of all options
system("--");
↦ // --batch           0
↦ // --execute
↦ // --sdb             0
↦ // --echo            1
↦ // --profile         0
↦ // --quiet           1
↦ // --sort            0
```

```
↦ // --random          12345678
↦ // --no-tty          1
↦ // --user-option
↦ // --allow-net       0
↦ // --browser
↦ // --cntrlc
↦ // --emacs           0
↦ // --no-stdlib       0
↦ // --no-rc           1
↦ // --no-warn         0
↦ // --no-out          0
↦ // --no-shell        0
↦ // --min-time        "0.5"
↦ // --cpus            4
↦ // --threads         4
↦ // --MPport
↦ // --MPhost
↦ // --link
↦ // --ticks-per-sec   1
// set minimal display time to 0.02 seconds
system("--min-time", "0.02");
// set timer resolution to 0.01 seconds
system("--ticks-per-sec", 100);
// re-seed random number generator
system("--random", 12345678);
// allow your web browser to access HTML pages from the net
system("--allow-net", 1);
// and set help browser to firefox
system("--browser", "firefox");
↦ // ** Could not get 'DataDir'.
↦ // ** Either set environment variable 'SINGULAR_DATA_DIR' to 'DataDir',
↦ // ** or make sure that 'DataDir' is at "/home/hannes/singular/doc/../Sin\
   gular/../share/"
↦ // ** Could not get 'IdxFile'.
↦ // ** Either set environment variable 'SINGULAR_IDX_FILE' to 'IdxFile',
↦ // ** Could not get 'DataDir'.
↦ // ** Either set environment variable 'SINGULAR_DATA_DIR' to 'DataDir',
↦ // ** or make sure that 'DataDir' is at "/home/hannes/singular/doc/../Sin\
   gular/../share/"
↦ // ** or make sure that 'IdxFile' is at "%D/singular/singular.idx"
↦ // ** resource `x` not found
↦ // ** Setting help browser to 'dummy'.
```

5.1.152 syz

Syntax: `syz` (ideal_expression)
`syz` (module_expression)
`syz` (ideal_expression, string_expression)
`syz` (module_expression, string_expression)

Type: module

Purpose: computes the first syzygy (i.e., the module of relations of the given generators) of the ideal, resp. module.

An optional second argument specifies the Groebner base algorith to use. Possible values are `"std"`(default) and `"slimgb"`.
Only for use of "std": If `option(returnSB)` is set, a standard basis is returned, otherwise a generating set.

Example:

```
  ring R=0,(x,y),(c,dp);
  ideal i=x,y;
  syz(i);
↦ _[1]=[y,-x]
```

See Section 5.1.46 [fres], page 185; Section 5.1.56 [hres], page 195; Section 4.5 [ideal], page 79; Section 5.1.78 [lift], page 209; Section 5.1.79 [liftstd], page 209; Section 5.1.81 [lres], page 212; Section 4.13 [module], page 111; Section 5.1.96 [mres], page 223; Section D.15.29 [nfmodsyz_lib], page 859; Section 5.1.103 [nres], page 228; Section 5.1.108 [option], page 230; [res], page 721; Section 5.1.145 [sres], page 264.

5.1.153 trace

Syntax: `trace` (intmat_expression)
`trace` (matrix_expression)

Type: int, if the argument is an intmat, resp.
poly, if the argument is a matrix

Purpose: returns the trace of an intmat, resp. matrix.

Example:

```
  intmat m[2][2]=1,2,3,4;
  print(m);
↦      1     2
↦      3     4
  trace(m);
↦ 5
```

See Section 4.7 [intmat], page 89; Section 4.12 [matrix], page 107.

5.1.154 transpose

Syntax: `transpose` (intmat_expression)
`transpose` (matrix_expression)
`transpose` (module_expression)

Type: intmat, matrix, or module, corresponding to the argument

Purpose: transposes a matrix.

Example:

```
  ring R=0,x,dp;
  matrix m[2][3]=1,2,3,4,5,6;
  print(m);
↦ 1,2,3,
↦ 4,5,6
  print(transpose(m));
↦ 1,4,
↦ 2,5,
↦ 3,6
```

See Section 4.7 [intmat], page 89; Section 4.12 [matrix], page 107; Section 4.13 [module], page 111.

5.1.155 type

Syntax: `type` name ;
`type (` name `);`

Type: none

Purpose: prints the name, level, type and value of a variable. To display the value of an expression, it is sufficient to type the expression followed by `;`.

Example:

```
  int i=3;
  i;
↦ 3
  type(i);
↦ // i int 3
```

See Chapter 4 [Data types], page 73; Section 5.1.80 [listvar], page 210; Section 5.1.117 [print], page 238.

5.1.156 typeof

Syntax: `typeof (` expression `)`

Type: string

Purpose: returns the type of an expression as string.

Returns the type of the first list element if the expression is an expression list.

Possible types are: `"ideal"`, `"int"`, `"intmat"`, `"intvec"`, `"list"`, `"map"`, `"matrix"`, `"module"`, `"number"`, `"none"`, `"poly"`, `"proc"`, `"qring"`, `"resolution"`, `"ring"`, `"string"`, `"vector"`.

For internal use only is the type `"?unknown type?"`.

Example:

```
  int i=9; i;
↦ 9
  typeof(_);
↦ int
  print(i);
↦ 9
  typeof(_);
↦ string
  type i;
↦ // i int 9
  typeof(_);
↦ string
  string s=typeof(i);
  s;
↦ int
  typeof(s);
↦ string
  proc p() {  "hello"; return();}
```

```
  p();
↦ hello
  typeof(_);
↦ ?unknown type?
```

See Chapter 4 [Data types], page 73; Section 5.1.155 [type], page 277.

5.1.157 univariate

Syntax: `univariate` (poly_expression)

Type: int

Purpose: returns 0 for not univariate, -1 for a constant or the number of the variable of the univariate polynomial.

Example:

```
  ring r=0,(x,y,z),dp;
  univariate(x2+1);
↦ 1
  univariate(x2+y+1);
↦ 0
  univariate(1);
↦ -1
  univariate(var(2));
↦ 2
  var(univariate(z));
↦ z
```

See Section 5.1.75 [leadexp], page 207; Section 5.1.160 [var], page 279.

5.1.158 uressolve

Syntax: `uressolve` (ideal_expression`,` int_expression`,` int_expression`,` int_expression)

Type: list

Purpose: computes all complex roots of a zerodimensional ideal.
Makes either use of the multipolynomial resultant of Macaulay (second argument = 1), which works only for homogeneous ideals, or uses the sparse resultant of Gelfand, Kapranov and Zelevinsky (second argument = 0).
The sparse resultant algorithm uses a mixed polyhedral subdivision of the Minkowski sum of the Newton polytopes in order to construct the sparse resultant matrix. Its determinant is a nonzero multiple of the sparse resultant. The u-resultant of B.\ L. van der Waerden and Laguerre's algorithm are used to determine the complex roots.
The third argument defines the precision of the fractional part if the ground field is the field of rational numbers, otherwise it will be ignored.
The fourth argument (can be 0, 1 or 2) gives the number of extra runs of Laguerre's algorithm (with corrupted roots), leading to better results.

Note: If the ground field is the field of complex numbers, the elements of the list are of type number, otherwise of type string.

See Section 5.1.72 [laguerre], page 206; Section 5.1.95 [mpresmat], page 222.

5.1.159 vandermonde

Syntax: `vandermonde` (ideal_expression, ideal_expression, int_expression)

Type: poly

Purpose: `vandermonde(p,v,d)` computes the (unique) polynomial of degree `d` with prescribed values `v[1],...,v[N]` at the points $\mathtt{p}^0,\ldots,\mathtt{p}^{N-1}$, `N=(d+1)`^{n}, n the number of ring variables.

The returned polynomial is $\sum c_{\alpha_1\ldots\alpha_n}\cdot x_1^{\alpha_1}\cdot\ldots\cdot x_n^{\alpha_n}$, where the coefficients $c_{\alpha_1\ldots\alpha_n}$ are the solution of the (transposed) Vandermonde system of linear equations

$$\sum_{\alpha_1+\ldots+\alpha_n\leq d} c_{\alpha_1\ldots\alpha_n}\cdot \mathtt{p}_1^{(k-1)\alpha_1}\cdot\ldots\cdot\mathtt{p}_n^{(k-1)\alpha_n} = \mathtt{v}[k], \quad k=1,\ldots,\mathtt{N}.$$

Note: the ground field has to be the field of rational numbers. Moreover, `ncols(p)==`n, the number of variables in the basering, and all the given generators have to be numbers different from 0,1 or -1. Finally, `ncols(v)==(d+1)`^{n}, and all given generators have to be numbers.

Example:

```
ring r=0,(x,y),dp;
// determine f with deg(f)=2 and with given values v of f
// at 9 points: (2,3)^0=(1,1),...,(2,3)^8=(2^8,3^8)
// valuation point: (2,3)
ideal p=2,3;
ideal v=1,2,3,4,5,6,7,8,9;
poly ip=vandermonde(p,v,2);
ip[1..5];  //  the 5 first terms of ip:
↦ -1/9797760x2y2-595/85536x2y+55/396576xy2+935/384x2-1309/3240xy
// compute value of ip at the point 2^8,3^8, result must be 9
subst(subst(ip,x,2^8),y,3^8);
↦ 9
```

5.1.160 var

Syntax: `var` (int_expression)

Type: poly

Purpose: `var(n)` returns the n-th ring variable.

Example:

```
  ring r=0,(x,y,z),dp;
  var(2);
↦ y
```

See Section 4.6 [int], page 83; Section 5.1.106 [nvars], page 230; Section 4.19 [ring], page 125; Section 5.1.135 [rvar], page 254; Section 5.1.157 [univariate], page 278; Section 5.1.162 [varstr], page 280.

5.1.161 variables

Syntax: `variables` (poly_expression)
`variables` (ideal_expression)
`variables` (matrix_expression)

Type: ideal

Purpose: `variables(p)` returns the list of all ring variables the argument depends on.

Example:

```
  ring r=0,(x,y,z),dp;
  variables(2);
↦ _[1]=0
  variables(x+y2);
↦ _[1]=x
↦ _[2]=y
  variables(ideal(x+y2,x3y,z));
↦ _[1]=x
↦ _[2]=y
↦ _[3]=z
  string(variables(ideal(x+y2,x3y,z)));
↦ x,y,z
```

See Section 5.1.75 [leadexp], page 207; Section 5.1.106 [nvars], page 230; Section 5.1.157 [univariate], page 278; Section 5.1.160 [var], page 279; Section 5.1.162 [varstr], page 280.

5.1.162 varstr

Syntax: `varstr` (ring_name)
`varstr` (int_expression)
`varstr` (ring_name`,` int_expression)

Type: string

Purpose: returns the list of the names of the ring variables as a string or the name of the n-th ring variable, where n is given by the int_expression.
If the ring name is omitted, the basering is used, thus `varstr(n)` is equivalent to `varstr(basering,n)`.

Example:

```
  ring r=0,(x,y,z),dp;
  varstr(r);
↦ x,y,z
  varstr(r,1);
↦ x
  varstr(2);
↦ y
```

See Section 5.1.7 [charstr], page 160; Section 4.6 [int], page 83; Section 5.1.106 [nvars], page 230; Section 5.1.110 [ordstr], page 235; Section 5.1.113 [parstr], page 236; Section 4.19 [ring], page 125; Section 5.1.160 [var], page 279.

5.1.163 vdim

Syntax: `vdim` (ideal_expression)
`vdim` (module_expression)

Type: int

Purpose: computes the vector space dimension of the ring, resp. free module, modulo the ideal, resp. module, generated by the initial terms of the given generators. If the generators

form a standard basis, this is the same as the vector space dimension of the ring, resp. free module, modulo the ideal, resp. module.
If the ideal, resp. module, is not zero-dimensional, -1 is returned.

Example:

```
  ring r=0,(x,y),ds;
  ideal i=x2+y2,x2-y2;
  ideal j=std(i);
  vdim(j);
↦ 4
```

See [codim], page 783; Section 5.1.19 [degree], page 167; Section 5.1.24 [dim], page 170; Section 4.5 [ideal], page 79; Section 5.1.67 [kbase], page 203; Section 5.1.98 [mult], page 224; Section 5.1.147 [std], page 266.

5.1.164 waitall

Syntax: `waitall` (list_expression)
`waitall` (list_expression `,` int_expression)

Type: int

Purpose: Expects a list of open links (of mode ssi:fork, ssi:tcp) and waits until all of them are finished, i.e., are ready to be read.
In the first case, the command waits for all links to finish (or to crash, see below) and may therefore run forever.
In the second case, a timeout in milliseconds can be provided, forcing the command to terminate after the specified time. If the given timeout is 0, the command checks whether all links are finished or not, but does not wait for any link (polling).
Return values are:
-1: The read state of all links is `"eof"`, see Section 4.9 [link], page 95, Section 5.1.146 [status], page 265. This might happen if all the links crashed.
0: timeout (or polling): None of the links is ready.
1: All links are ready. (Note: There might be links whose read state is `"eof"`, but at least one link is ready.)

Example:

```
  link l1 = "ssi:fork"; open(l1);
  link l2 = "ssi:fork"; open(l2);
  link l3 = "ssi:fork"; open(l3);
  list l = list(l1,l2,l3);
  write(l1, quote(system("sh", "sleep 15")));
  write(l2, quote(system("sh", "sleep 10")));
  write(l3, quote(system("sh", "sleep 11")));
  waitall(l, 5000); // terminates after 5sec with result 0
↦ 0
  waitall(l);       // terminates after 10 more sec
↦ 1
  close(l1);
  close(l2);
  close(l3);
```

See Section 5.1.165 [waitfirst], page 282.

5.1.165 waitfirst

Syntax: `waitfirst` (list_expression)
`waitfirst` (list_expression `,` int_expression)

Type: int

Purpose: Expects a list of open links (of mode ssi:fork, ssi:tcp) and waits until the first of them is finished, i.e., is ready to be read.
In the first case, the command waits until the first link is finished (or all of them crashed, see below) and may therefore run forever.
In the second case, a timeout in milliseconds can be provided, forcing the command to terminate after the specified time. If the given timeout is 0, the command checks whether one of the links is finished or not, but does not wait for any link (polling).
Return values are:
-1: The read state of all links is `"eof"`, see Section 4.9 [link], page 95, Section 5.1.146 [status], page 265. This might happen if all the links crashed.
0: timeout (or polling): None of the links is ready.
i>1: At least the link whose list index is i is ready.

Example:

```
  link l1 = "ssi:fork"; open(l1);
  link l2 = "ssi:fork"; open(l2);
  link l3 = "ssi:fork"; open(l3);
  list l = list(l1,l2,l3);
  write(l1, quote(system("sh", "sleep 15")));
  write(l2, quote(system("sh", "sleep 13")));
  write(l3, quote(system("sh", "sleep 11")));
  waitfirst(l, 5000); // terminates after 5sec with result 0
↦ 0
  waitfirst(l);       // terminates after 6 more sec with result 3
↦ 3
  close(l1);
  close(l2);
  close(l3);
```

See Section 5.1.164 [waitall], page 281.

5.1.166 wedge

Syntax: `wedge` (matrix_expression`,` int_expression)

Type: matrix

Purpose: `wedge(M,n)` computes the `n`-th exterior power of the matrix `M`.

Example:

```
  ring r;
  matrix m[2][3]=x,y,y,z,z,x;
  print(m);
↦ x,y,y,
↦ z,z,x
  print(wedge(m,2));
↦ xz-yz,-x2+yz,xy-yz
```

See Section 4.6 [int], page 83; Section 4.12 [matrix], page 107; Section 5.1.90 [minor], page 218.

5.1.167 weight

Syntax: `weight` (ideal_expression)
`weight` (module_expression)

Type: intvec

Purpose: computes an "optimal" weight vector for an ideal, resp. module, which may be used as weight vector for the variables in order to speed up the standard basis algorithm. If the input is weighted homogeneous, a weight vector for which the input is weighted homogeneous is found.

Example:

```
ring h1=32003,(t,x,y,z),dp;
ideal i=
9x8+y7t3z4+5x4y2t2+2xy2z3t2,
9y8+7xy6t+2x5y4t2+2x2yz3t2,
9z8+3x2y3z2t4;
intvec e=weight(i);
e;
↦ 5,7,5,7
ring r=32003,(a,b,c,d),wp(e);
map f=h1,a,b,c,d;
ideal i0=std(f(i));
```

See Section 4.5 [ideal], page 79; Section 4.8 [intvec], page 92; Section 5.1.120 [qhweight], page 242.

5.1.168 weightKB

Procedure from library `standard.lib` (see Section D.1 [standard_lib], page 721).

Syntax: `weightKB` (module_expression, int_expression , list_expression)
`weightKB` (ideal_expression, int_expression, list_expression)

Return: the same as the input type of the first argument

Purpose: If `I,d,wim` denotes the three arguments then weightKB computes the weighted degree-`d` part of a vector space basis (consisting of monomials) of the quotient ring, resp. of the quotient module, modulo `I` w.r.t. weights given by `wim` The information about the weights is given as a list of two intvec: `wim[1]` weights for all variables (positive), `wim[2]` weights for the module generators.

Note: This is a generalization of the command `kbase` with the same first two arguments.

Example:

```
  ring R=0, (x,y), wp(1,2);
weightKB(ideal(0),3,intvec(1,2));
↦ _[1]=x3
↦ _[2]=xy
```

See also: Section 5.1.67 [kbase], page 203.

5.1.169 write

Syntax: `write` (link_expression, expression_list)
for DBM links:
`write` (link, string_expression, string_expression)
`write` (link, string_expression)

Type: none

Purpose: writes data to a link.
If the link is of type `ASCII`, all expressions are converted to strings (and separated by a newline character) before they are written. As a consequence, only such values which can be converted to a string can be written to an `ASCII` link.
For ssi links, ring-dependent expressions are written together with a ring description. To prevent an evaluation of the expression before it is written, the `quote` command (possibly together with `eval`) can be used. A `write` blocks (i.e., does not return to the prompt), as long as a ssi link is not ready for writing.
For DBM links, `write` with three arguments inserts the first string as key and the second string as value into the dbm data base.
Called with two arguments, it deletes the entry with the key specified by the string from the data base.

Example:

```
// write the values of the variables f and i as strings into
// the file "outfile" (overwrite it, if it exists)
write(":w outfile",f,i);

// now append the string "that was f,i" (without the quotes)
// at the end of the file "outfile"
write(":a outfile","that was f,i");
// alternatively, links could be used:
link l=":a outfile"; l;
// type : ASCII
// mode : a
// name : outfile
// open : no
// read : not ready
// write: not ready
write(l," that was f,i");
// saving and retrieving data (ASCII format):
ring r=32003,(x,y,z),dp;
ideal i=x+y,z3+22y;
write(":w save_i",i);// this writes x+y,z3+22y to the file save_i
ring r=32003,(x,y,z),dp;
string s=read("save_i");   //creates the string x+y,z3+22y
execute("ideal k="+s+";"); // this defines an ideal k which
                           // is equal to i.
// for large objects, the ssi format and ssi links are better:
write("ssi:w save_i.ssi",i);
def j=read("ssi:r save_i.ssi");
```

See Chapter 4 [Data types], page 73; Section 5.1.26 [dump], page 172; Section 5.1.28 [eval], page 174; Section 4.9 [link], page 95; Section 5.1.117 [print], page 238; [printf], page 721; Section 5.1.122 [quote], page 243; Section 5.1.126 [read], page 245; Section 5.3.7 [short], page 299.

5.2 Control structures

A sequence of commands surrounded by curly brackets (`{` and `}`) is a so-called block. Blocks are used in SINGULAR in order to define procedures and to collect commands belonging to `if`, `else`, `for` and `while` statements and to the `example` part in libraries. Even if the sequence of statements consists of only a single command it has to be surrounded by curly brackets! Variables which are defined inside a block are not local to that block. Note that there need not be an ending semicolon at the end of the block.

Example:

```
if ( i>j )
{
  // This is the block
  int temp;
  temp=i;
  i=j;
  j=temp;
  kill temp;
}
```

5.2.1 apply

Syntax: `apply(` expression , function `);`

Purpose: applies the function to all elements of the first argument. The first argument must be of type `intvec`, `intmat`, or `list`. The result will be an expression list, its type and format will be set by the following assign. The function must be a kernel command or a procedure which takes one argument and returns a a value.

Example:

```
proc p(int x) {return(x^2);}
intvec v=1,2,3;
apply(v,p);
↦ 1 4 9
intvec vv=apply(v,p);vv;
↦ 1,4,9
list ll=apply(v,p);ll;
↦ [1]:
↦    1
↦ [2]:
↦    4
↦ [3]:
↦    9
```

5.2.2 break

Syntax: `break;`

Purpose: leaves the innermost `for` or `while` block.

Example:

```
while (1)
{
  ...
  if ( ... )
  {
    break; // leave the while block
  }
}
```

See Section 5.2 [Control structures], page 284; Section 5.2.8 [for], page 290; Section 5.2.15 [while], page 296.

5.2.3 breakpoint

Syntax: `breakpoint(` proc_name `);`
`breakpoint(` proc_name`,` line_no `);`

Purpose: sets a breakpoint at the beginning of the specified procedure or at the given line.
Note: Line number 1 is the first line of a library (for procedures from libraries), resp. the line with the `{`.
A line number of -1 removes all breakpoint from that procedure.

Example:

```
breakpoint(groebner);
↦ breakpoint 1, at line 838 in groebner
breakpoint(groebner, 176);
↦ breakpoint 2, at line 176 in groebner
breakpoint(groebner, -1);
↦ breakpoints in groebner deleted(0x6)
```

See Section 3.9.3 [Source code debugger], page 69; Section 5.2.16 [~], page 296.

5.2.4 continue

Syntax: `continue;`

Purpose: skips the rest of the innermost `for` or `while` loop und jumps to the beginning of the block. This command is only valid inside a `for` or a `while` construction.

Note: Unlike the C-construct it **does not execute the increment statement**. The command `continue` is mainly for internal use.

Example:

```
for (int i = 1 ; i<=10; i=i+1)
{
   ...
   if (i==3) { i=8;continue; }
     // skip the rest if i is 3 and
     // continue with the next i: 8
   i;
}
↦ 1
↦ 2
↦ 8
↦ 9
↦ 10
```

See Section 5.2 [Control structures], page 284; Section 5.2.8 [for], page 290; Section 5.2.15 [while], page 296.

5.2.5 else

Syntax: `if (` boolean_expression `)` true_block `else` false_block

Purpose: executes false_block if the boolean_expression of the `if` statement is false. This command is only valid in combination with an `if` command.

Example:

```
int i=3;
if (i > 5)
{
  "i is bigger than 5";
}
else
{
  "i is smaller than 6";
}
↦ i is smaller than 6
```

See Section 5.2 [Control structures], page 284; Section 4.6.5 [boolean expressions], page 87; Section 5.2.9 [if], page 290.

5.2.6 export

Syntax: `export` name `;`
`export` list_of_names `;`

Purpose: converts a local variable of a procedure to a global one, that is the identifier is not removed at the end of the procedure. However, the package the variable belongs to is not changed.

Note: Objects defined in a ring are not automatically exported when exporting the ring.

Example:

```
proc p1
{
  int i,j;
  export(i);
  intmat m;
  listvar();
  export(m);
}
p1();
↦ // m                    [1]  intmat 1 x 1
↦ // j                    [1]  int 0
↦ // i                    [0]  int 0
listvar();
↦ // m                    [0]  intmat 1 x 1
↦ // i                    [0]  int 0
```

See Section 5.2.7 [exportto], page 287; Section 5.2.10 [importfrom], page 291; Section 5.2.11 [keepring], page 293.

5.2.7 exportto

Syntax: `exportto(` package_name `,` name `);`
`exportto(` package_name `,` list_of_names `);`

Purpose: transfers an identifier in the current package into the one specified by package_name. package_name can be `Current`, `Top` or any other identifier of type package.

Note: Objects defined in a ring are not automatically exported when exporting the ring.

Warning: The identifier is transfered to the other package. It does no longer exist in the current package. If the identifier should only be copied, Section 5.2.10 [importfrom], page 291 should be used instead.

Example:

```
proc p1
{
  int i,j;
  exportto(Current,i);
  intmat m;
  listvar(Current);
  exportto(Top,m);
}
p1();
↦ // Top                          [0]  package Top (T)
↦ // ::m                          [1]  intmat 1 x 1
↦ // ::i                          [0]  int 0
↦ // ::j                          [1]  int 0
↦ // ::#                          [1]  list, size: 0
↦ // ::p1                         [0]  proc
↦ // ::mathicgb_prOrder           [0]  proc from singmathic.so (C)
↦ // ::mathicgb                   [0]  proc from singmathic.so (C)
↦ // ::min                        [0]  proc from standard.lib
↦ // ::max                        [0]  proc from standard.lib
↦ // ::datetime                   [0]  proc from standard.lib
↦ // ::weightKB                   [0]  proc from standard.lib
↦ // ::fprintf                    [0]  proc from standard.lib
↦ // ::printf                     [0]  proc from standard.lib
↦ // ::sprintf                    [0]  proc from standard.lib
↦ // ::quotient4                  [0]  proc from standard.lib
↦ // ::quotient5                  [0]  proc from standard.lib
↦ // ::quotient3                  [0]  proc from standard.lib
↦ // ::quotient2                  [0]  proc from standard.lib
↦ // ::quotient1                  [0]  proc from standard.lib
↦ // ::quot                       [0]  proc from standard.lib
↦ // ::res                        [0]  proc from standard.lib
↦ // ::groebner                   [0]  proc from standard.lib
↦ // ::qslimgb                    [0]  proc from standard.lib
↦ // ::hilbRing                   [0]  proc from standard.lib
↦ // ::par2varRing                [0]  proc from standard.lib
↦ // ::quotientList               [0]  proc from standard.lib
↦ // ::stdhilb                    [0]  proc from standard.lib
↦ // ::stdfglm                    [0]  proc from standard.lib
↦ // ::flintZ                     [0]  proc from kernel (C)
↦ // ::Float                      [0]  proc from kernel (C)
↦ // ::crossprod                  [0]  proc from kernel (C)
package Test1;
exportto(Test1,p1);
listvar(Top);
↦ // Top                          [0]  package Top (T)
↦ // ::m                          [0]  intmat 1 x 1
↦ // ::i                          [0]  int 0
```

```
↦ // ::mathicgb_prOrder            [0]  proc from singmathic.so (C)
↦ // ::mathicgb                    [0]  proc from singmathic.so (C)
↦ // ::min                         [0]  proc from standard.lib
↦ // ::max                         [0]  proc from standard.lib
↦ // ::datetime                    [0]  proc from standard.lib
↦ // ::weightKB                    [0]  proc from standard.lib
↦ // ::fprintf                     [0]  proc from standard.lib
↦ // ::printf                      [0]  proc from standard.lib
↦ // ::sprintf                     [0]  proc from standard.lib
↦ // ::quotient4                   [0]  proc from standard.lib
↦ // ::quotient5                   [0]  proc from standard.lib
↦ // ::quotient3                   [0]  proc from standard.lib
↦ // ::quotient2                   [0]  proc from standard.lib
↦ // ::quotient1                   [0]  proc from standard.lib
↦ // ::quot                        [0]  proc from standard.lib
↦ // ::res                         [0]  proc from standard.lib
↦ // ::groebner                    [0]  proc from standard.lib
↦ // ::qslimgb                     [0]  proc from standard.lib
↦ // ::hilbRing                    [0]  proc from standard.lib
↦ // ::par2varRing                 [0]  proc from standard.lib
↦ // ::quotientList                [0]  proc from standard.lib
↦ // ::stdhilb                     [0]  proc from standard.lib
↦ // ::stdfglm                     [0]  proc from standard.lib
↦ // ::flintZ                      [0]  proc from kernel (C)
↦ // ::Float                       [0]  proc from kernel (C)
↦ // ::crossprod                   [0]  proc from kernel (C)
listvar(Test1);
↦ // Test1                         [0]  package Test1 (N)
↦ // ::p1                          [0]  proc
Test1::p1();
↦ // Test1                         [0]  package Test1 (N)
↦ // ::m                           [1]  intmat 1 x 1
↦ // ::i                           [0]  int 0
↦ // ::j                           [1]  int 0
↦ // ::#                           [1]  list, size: 0
↦ // ::p1                          [0]  proc
↦ // ** redefining m (  exportto(Top,m);)
listvar(Top);
↦ // Top                           [0]  package Top (T)
↦ // ::m                           [0]  intmat 1 x 1
↦ // ::i                           [0]  int 0
↦ // ::mathicgb_prOrder            [0]  proc from singmathic.so (C)
↦ // ::mathicgb                    [0]  proc from singmathic.so (C)
↦ // ::min                         [0]  proc from standard.lib
↦ // ::max                         [0]  proc from standard.lib
↦ // ::datetime                    [0]  proc from standard.lib
↦ // ::weightKB                    [0]  proc from standard.lib
↦ // ::fprintf                     [0]  proc from standard.lib
↦ // ::printf                      [0]  proc from standard.lib
↦ // ::sprintf                     [0]  proc from standard.lib
↦ // ::quotient4                   [0]  proc from standard.lib
↦ // ::quotient5                   [0]  proc from standard.lib
↦ // ::quotient3                   [0]  proc from standard.lib
```

```
↦ // ::quotient2                  [0]  proc from standard.lib
↦ // ::quotient1                  [0]  proc from standard.lib
↦ // ::quot                       [0]  proc from standard.lib
↦ // ::res                        [0]  proc from standard.lib
↦ // ::groebner                   [0]  proc from standard.lib
↦ // ::qslimgb                    [0]  proc from standard.lib
↦ // ::hilbRing                   [0]  proc from standard.lib
↦ // ::par2varRing                [0]  proc from standard.lib
↦ // ::quotientList               [0]  proc from standard.lib
↦ // ::stdhilb                    [0]  proc from standard.lib
↦ // ::stdfglm                    [0]  proc from standard.lib
↦ // ::flintZ                     [0]  proc from kernel (C)
↦ // ::Float                      [0]  proc from kernel (C)
↦ // ::crossprod                  [0]  proc from kernel (C)
listvar(Test1);
↦ // Test1                        [0]  package Test1 (N)
↦ // ::i                          [0]  int 0
↦ // ::p1                         [0]  proc
```

See Section 5.2.6 [export], page 287; Section 5.2.10 [importfrom], page 291; Section 5.2.11 [keepring], page 293.

5.2.8 for

Syntax: `for (` init_command`;` boolean_expression`;` iterate_commands`)` block

Purpose: repetitive, conditional execution of a command block.
The command init_command is executed first. Then boolean_expression is evaluated. If its value is TRUE the block is executed, otherwise the `for` statement is complete. After each execution of the block, the command iterate_command is executed and boolean_expression is evaluated. This is repeated until boolean_expression evaluates to FALSE.
The command `break;` leaves the innermost `for` construct.

Example:

```
// sum of 1 to 10:
int s=0;
for (int i=1; i<=10; i=i+1)
{
   s=s+i;
}
s;
↦ 55
```

See Section 5.2 [Control structures], page 284; Section 4.6.5 [boolean expressions], page 87; Section 5.2.2 [break], page 285; Section 5.2.4 [continue], page 286; Section 5.2.9 [if], page 290; Section 5.2.15 [while], page 296.

5.2.9 if

Syntax: `if (` boolean_expression `)` true_block
`if (` boolean_expression `)` true_block `else` false_block

Purpose: executes true_block if the boolean condition is true. If the `if` statement is followed by an `else` statement and the boolean condition is false, then false_block is executed.

Example:

```
int i = 9;
matrix m[i][i];
if (i > 5 and typeof(m) == "matrix")
{
  m[i][i] = i;
}
```

See Section 5.2 [Control structures], page 284; Section 4.6.5 [boolean expressions], page 87; Section 5.2.2 [break], page 285; Section 5.2.5 [else], page 286.

5.2.10 importfrom

Syntax: `importfrom(` package_name `,` name `);`
`importfrom(` package_name `,` list_of_names `);`

Purpose: creates a new identifier in the current package which is a copy of the one specified by name in the package package_name. package_name can be `Top` or any other identifier of type package.

Note: Objects defined in a ring are not automatically imported when importing the ring.

Warning: The identifier is copied to the current package. It does still exist (independently) in the package package_name. If the identifier should be erased in the package from which it originates, Section 5.2.7 [exportto], page 287 should be used instead.

Example:

```
listvar(Top);
↦ // Top                           [0]  package Top (T)
↦ // ::mathicgb_prOrder            [0]  proc from singmathic.so (C)
↦ // ::mathicgb                    [0]  proc from singmathic.so (C)
↦ // ::min                         [0]  proc from standard.lib
↦ // ::max                         [0]  proc from standard.lib
↦ // ::datetime                    [0]  proc from standard.lib
↦ // ::weightKB                    [0]  proc from standard.lib
↦ // ::fprintf                     [0]  proc from standard.lib
↦ // ::printf                      [0]  proc from standard.lib
↦ // ::sprintf                     [0]  proc from standard.lib
↦ // ::quotient4                   [0]  proc from standard.lib
↦ // ::quotient5                   [0]  proc from standard.lib
↦ // ::quotient3                   [0]  proc from standard.lib
↦ // ::quotient2                   [0]  proc from standard.lib
↦ // ::quotient1                   [0]  proc from standard.lib
↦ // ::quot                        [0]  proc from standard.lib
↦ // ::res                         [0]  proc from standard.lib
↦ // ::groebner                    [0]  proc from standard.lib
↦ // ::qslimgb                     [0]  proc from standard.lib
↦ // ::hilbRing                    [0]  proc from standard.lib
↦ // ::par2varRing                 [0]  proc from standard.lib
↦ // ::quotientList                [0]  proc from standard.lib
↦ // ::stdhilb                     [0]  proc from standard.lib
↦ // ::stdfglm                     [0]  proc from standard.lib
↦ // ::flintZ                      [0]  proc from kernel (C)
↦ // ::Float                       [0]  proc from kernel (C)
↦ // ::crossprod                   [0]  proc from kernel (C)
```

```
load("inout.lib");
listvar(Top);
↦ // Top                        [0]  package Top (T)
↦ // ::mathicgb_prOrder         [0]  proc from singmathic.so (C)
↦ // ::mathicgb                 [0]  proc from singmathic.so (C)
↦ // ::min                      [0]  proc from standard.lib
↦ // ::max                      [0]  proc from standard.lib
↦ // ::datetime                 [0]  proc from standard.lib
↦ // ::weightKB                 [0]  proc from standard.lib
↦ // ::fprintf                  [0]  proc from standard.lib
↦ // ::printf                   [0]  proc from standard.lib
↦ // ::sprintf                  [0]  proc from standard.lib
↦ // ::quotient4                [0]  proc from standard.lib
↦ // ::quotient5                [0]  proc from standard.lib
↦ // ::quotient3                [0]  proc from standard.lib
↦ // ::quotient2                [0]  proc from standard.lib
↦ // ::quotient1                [0]  proc from standard.lib
↦ // ::quot                     [0]  proc from standard.lib
↦ // ::res                      [0]  proc from standard.lib
↦ // ::groebner                 [0]  proc from standard.lib
↦ // ::qslimgb                  [0]  proc from standard.lib
↦ // ::hilbRing                 [0]  proc from standard.lib
↦ // ::par2varRing              [0]  proc from standard.lib
↦ // ::quotientList             [0]  proc from standard.lib
↦ // ::stdhilb                  [0]  proc from standard.lib
↦ // ::stdfglm                  [0]  proc from standard.lib
↦ // ::flintZ                   [0]  proc from kernel (C)
↦ // ::Float                    [0]  proc from kernel (C)
↦ // ::crossprod                [0]  proc from kernel (C)
importfrom(Inout,pause);
listvar(Top);
↦ // Top                        [0]  package Top (T)
↦ // ::pause                    [0]  proc from inout.lib
↦ // ::mathicgb_prOrder         [0]  proc from singmathic.so (C)
↦ // ::mathicgb                 [0]  proc from singmathic.so (C)
↦ // ::min                      [0]  proc from standard.lib
↦ // ::max                      [0]  proc from standard.lib
↦ // ::datetime                 [0]  proc from standard.lib
↦ // ::weightKB                 [0]  proc from standard.lib
↦ // ::fprintf                  [0]  proc from standard.lib
↦ // ::printf                   [0]  proc from standard.lib
↦ // ::sprintf                  [0]  proc from standard.lib
↦ // ::quotient4                [0]  proc from standard.lib
↦ // ::quotient5                [0]  proc from standard.lib
↦ // ::quotient3                [0]  proc from standard.lib
↦ // ::quotient2                [0]  proc from standard.lib
↦ // ::quotient1                [0]  proc from standard.lib
↦ // ::quot                     [0]  proc from standard.lib
↦ // ::res                      [0]  proc from standard.lib
↦ // ::groebner                 [0]  proc from standard.lib
↦ // ::qslimgb                  [0]  proc from standard.lib
↦ // ::hilbRing                 [0]  proc from standard.lib
↦ // ::par2varRing              [0]  proc from standard.lib
```

```
↦ // ::quotientList                  [0]  proc from standard.lib
↦ // ::stdhilb                       [0]  proc from standard.lib
↦ // ::stdfglm                       [0]  proc from standard.lib
↦ // ::flintZ                        [0]  proc from kernel (C)
↦ // ::Float                         [0]  proc from kernel (C)
↦ // ::crossprod                     [0]  proc from kernel (C)
```

See Section 5.2.6 [export], page 287; Section 5.2.7 [exportto], page 287; Section 5.2.11 [keepring], page 293.

5.2.11 keepring

Syntax: `keepring` name `;`

Warning: This command is obsolete. Instead the respective identifiers in the ring should be exported and the ring itself should subsequently be returned. The command is only included for backward compatibility and may be removed in future releases.

Purpose: moves the specified ring to the next (upper) level. This command can only be used inside of procedures and it should be the last command before the `return` statement. There it provides the possibility to keep a ring which is local to the procedure (and its objects) accessible after the procedure ended without making the ring global.

Example:

```
proc P1
{
  ring r=0,x,dp;
  keepring r;
}
proc P2
{
  "inside P2: " + nameof(basering);
  P1();
  "inside P2, after call of P1: " + nameof(basering);
}
ring r1= 0,y,dp;
P2();
↦ inside P2: r1
↦ inside P2, after call of P1: r
"at top level: " + nameof(basering);
↦ at top level: r1
```

See Section 4.19 [ring], page 125.

5.2.12 load

Syntax: `load(` string_expression `);`
`load(` string_expression `,` string_expression `);`

Type: none

Purpose: reads a library of procedures from a file. In contrast to the command `LIB` (see note below), the command `load` does not add the procedures of the library to the package `Top`, but only to the package corresponding to the library. If the given filename does not start with `.` or `/`, the following directories are searched for it (in the given order):

the current directory, the directories given in the environment variable `SINGULARPATH`, some default directories relative to the location of the SINGULAR executable program, and finally some default absolute directories. You can view the search path which SINGULAR uses to locate its libraries, by starting up SINGULAR with the option `-v`, or by issuing the command `system("with");`.

The second string selections options for loading.

Note: `load(<string_expr>,"with")` is equivalent to
`LIB <string_expr>`.

Note: `load(<string_expr>,"try")` is equivalent to
`LIB <string_expr>` which never fails - test the package name to distiguish.

All loaded libraries are displayed by the `listvar(package);` command:

```
  option(loadLib);   // show loading of libraries;
                     // standard.lib is loaded
  listvar(package);
↦ // Singmathic                     [0]  package Singmathic (C,singmathic.s\
   o)
↦ // Standard                       [0]  package Standard (S,standard.lib)
↦ // Top                            [0]  package Top (T)
                     // the names of the procedures of inout.lib
  load("inout.lib"); // are now known to Singular
↦ // ** loaded inout.lib (4.1.1.0,Dec_2017)
  listvar(package);
↦ // Inout                          [0]  package Inout (S,inout.lib)
↦ // Singmathic                     [0]  package Singmathic (C,singmathic.s\
   o)
↦ // Standard                       [0]  package Standard (S,standard.lib)
↦ // Top                            [0]  package Top (T)
  load("blabla.lib","try");
  listvar(package);
↦ // Inout                          [0]  package Inout (S,inout.lib)
↦ // Singmathic                     [0]  package Singmathic (C,singmathic.s\
   o)
↦ // Standard                       [0]  package Standard (S,standard.lib)
↦ // Top                            [0]  package Top (T)
  option(noloadLib);   // do not show loading of libraries;
  load("matrix.lib","try");
  listvar(package);
↦ // Elim                           [0]  package Elim (S,elim.lib)
↦ // Triang                         [0]  package Triang (S,triang.lib)
↦ // Absfact                        [0]  package Absfact (S,absfact.lib)
↦ // Primdec                        [0]  package Primdec (S,primdec.lib)
↦ // Ring                           [0]  package Ring (S,ring.lib)
↦ // Random                         [0]  package Random (S,random.lib)
↦ // General                        [0]  package General (S,general.lib)
↦ // Poly                           [0]  package Poly (S,poly.lib)
↦ // Nctools                        [0]  package Nctools (S,nctools.lib)
↦ // Matrix                         [0]  package Matrix (S,matrix.lib)
↦ // Inout                          [0]  package Inout (S,inout.lib)
↦ // Singmathic                     [0]  package Singmathic (C,singmathic.s\
   o)
↦ // Standard                       [0]  package Standard (S,standard.lib)
```

```
↦ // Top                            [0]  package Top (T)
```

Each time a library (Section 3.8 [Libraries], page 55) / dynamic module (Section 3.10 [Dynamic loading], page 72) is loaded, the corresponding package is created, if it does not already exist.

The name of a package corresponding to a SINGULAR library is derived from the name of the library file. The first letter is capitalized and everything to right of the left-most dot is dropped. For a dynamic module the packagename is hard-coded in the binary file.

Only the names of the procedures in the library are loaded, the body of the procedures is read during the first call of this procedure. This minimizes memory consumption by unused procedures. When SINGULAR is started with the `-q` or `--quiet` option, no message about the loading of a library is displayed.

```
  option(loadLib); // show loading of libraries; standard.lib is loaded
                   // the names of the procedures of inout.lib
  load("inout.lib"); // are now known to Singular
↦ // ** loaded inout.lib (4.1.1.0,Dec_2017)
  listvar();
```

See Section 3.1.6 [Command line options], page 19; Section A.1.9 [Dynamic modules], page 637; Section 5.1.77 [LIB], page 208; Section 2.3.3 [Procedures and libraries], page 10; Appendix D [SINGULAR libraries], page 721; Section 5.2.7 [exportto], page 287; Section 5.2.10 [importfrom], page 291; Section 4.15 [package], page 118; Section 4.17 [proc], page 122; Section D.1 [standard_lib], page 721; Section 4.20 [string], page 128; Section 5.1.151 [system], page 270.

5.2.13 quit

Syntax: `exit;`
`quit;`

Purpose: quits SINGULAR; works also from inside a procedure or from an interrupt. Instead of `quit`, the synonymous command `exit` may be used.

Example:

```
quit;
```

5.2.14 return

Syntax: `return` (expression_list `);`
`return ();`

Type: any

Purpose: returns the result(s) of a procedure and can only be used inside a procedure. Note that the brackets are required even if no return value is given.

Example:

```
proc p2
{
  int i,j;
  for(i=1;i<=10;i++)
  {
    j=j+i;
  }
  return(j);
}
```

```
// can also return an expression list, i.e., more than one value
proc tworeturn ()
{ return (1,2); }
int i,j = tworeturn();
// return type may even depend on the input
proc type_return (int i)
{
  if (i > 0) {return (i);}
  else {return (list(i));}
}
// then we need def type (or list) to collect value
def t1 = type_return(1);
def t2 = type_return(-1);
```

See Chapter 4 [Data types], page 73; Section 4.17 [proc], page 122.

5.2.15 while

Syntax: `while` (boolean_expression) block

Purpose: repetitive, conditional execution of block.
The boolean_expression is evaluated and if its value is TRUE, the block gets executed. This is repeated until boolean_expression evaluates to FALSE. The command `break` leaves the innermost `while` construction.

Example:

```
int i = 9;
while (i>0)
{
   // ... // do something for i=9, 8, ..., 1
   i = i - 1;
}
while (1)
{
   // ...   // do something forever
   if (i == -5) // but leave the loop if i is -5
   {
     break;
   }
}
```

See Section 5.2 [Control structures], page 284; Section 4.6.5 [boolean expressions], page 87; Section 5.2.2 [break], page 285.

5.2.16 ~ (break point)

Syntax: `~;`

Purpose: sets a break point. Whenever SINGULAR reaches the command `~;` in a sequence of commands it prompts for input. The user may now input lines of SINGULAR commands. The line length cannot exceed 80 characters. SINGULAR proceeds with the execution of the command following `~;` as soon as it receives an empty line.
Furthermore, the debug mode will be activated: See Section 3.9.3 [Source code debugger], page 69.

Example:

```
proc t
{
  int i=2;
  ~;
  return(i+1);
}
t();
↦ -- break point in t --
↦ -- 0: called    from STDIN --
// here local variables of the procedure can be accessed
i;
↦ 2
↦ -- break point in t --

↦ 3
```

See Section 3.9.4 [Break points], page 70.

5.3 System variables

5.3.1 degBound

Type: int

Purpose: The standard basis computation is stopped if the total (weighted) degree exceeds `degBound`.
`degBound` should not be used for a global ordering with inhomogeneous input, if the ordering is not `dp` or `Dp`. (Remark: elimination requires always an eliminiation ordering).
Reset this bound by setting `degBound` to 0.
The exact meaning of "degree" depends on the ring ordering and the command: `slimgb` uses always the total degree with weights 1, `std` does so for block orderings, only.

Example:

```
degBound = 7;
option();
↦ //options for 'std'-command: degBound
ideal j=std(i);
degBound;
↦ 7
degBound = 0; //resets degree bound to infinity
```

See Section 5.1.18 [deg], page 167; Section 4.6 [int], page 83; Section 5.1.108 [option], page 230; Section 5.1.147 [std], page 266.

5.3.2 echo

Type: int

Purpose: input is echoed if `echo >= voice`.
`echo` is a local setting for a procedure and defaulted to 0.
`echo` does not affect the output of commands.

Example:

```
echo = 1;
int i = echo;
↦ int i = echo;
```

See Section 4.6 [int], page 83; Section 5.3.11 [voice], page 303.

5.3.3 minpoly

Type: number

Purpose: describes the coefficient field of the current basering as an algebraic extension with the minimal polynomial equal to `minpoly`. Setting the `minpoly` should be the first command after defining the ring.

Note: The minimal polynomial has to be specified in the syntax of a polynomial. Its variable is not one of the ring variables, but the algebraic element which is being adjoined to the field. Algebraic extensions in SINGULAR are only possible over the rational numbers or over Z/p, p a prime number.
SINGULAR does not check whether the given polynomial is irreducible! It can be checked in advance with the function `factorize` (see Section 5.1.35 [factorize], page 178).

Example:

```
  //(Q[i]/(i^2+1))[x,y,z]:
  ring Cxyz=(0,i),(x,y,z),dp;
  minpoly=i^2+1;
  i2;  //this is a number, not a poly
↦ -1
```

See Section 5.1.35 [factorize], page 178; Section 4.19 [ring], page 125.

5.3.4 multBound

Type: int

Purpose: The standard basis computation is stopped if the ideal is zero-dimensional in a ring with local ordering and its multiplicity (`mult`) is lower than `multBound`.
Reset this bound by setting `multBound` to 0.

Example:

```
ring r=0,(x,y,z),ds;
ideal i,j;
i=x7+y7+z6,x6+y8+z7,x7+y5+z8,
x2y3+y2z3+x3z2,x3y2+y3z2+x2z3;
multBound=100;
j=std(i);
degree(j);
↦ // dimension (local)   = 0
↦ // multiplicity = 98
multBound=0;  //disables multBound
j=std(i);
degree(j);
↦ // dimension (local)   = 0
↦ // multiplicity = 86
```

See Section 4.6 [int], page 83; Section 5.1.98 [mult], page 224; Section 5.1.108 [option], page 230; Section 5.1.147 [std], page 266.

5.3.5 noether

Type: poly

Purpose: The standard basis computation in local rings cuts off all monomials above (in the sense of the monomial ordering) the monomial `noether` during the computation.
Reset `noether` by setting `noether` to 0.

Example:

```
  ring R=32003,(x,y,z),ds;
  ideal i=x2+y12,y13;
  std(i);
↦ _[1]=x2+y12
↦ _[2]=y13
  noether=x11;
  std(i);
↦ _[1]=x2
  noether=0; //disables noether
```

See Section 4.16 [poly], page 118; Section 5.1.147 [std], page 266.

5.3.6 printlevel

Type: int

Purpose: sets the debug level for `dbprint`. If `printlevel >= voice` then `dbprint` is equivalent to `print`, otherwise nothing is printed.

Note: See Section 3.8.6 [Procedures in a library], page 58, for a small example about how this is used for the display of comments while procedures are executed.

Example:

```
  voice;
↦ 1
  printlevel=0;
  dbprint(1);
  printlevel=voice;
  dbprint(1);
↦ 1
```

See Section 5.1.16 [dbprint], page 165; Section 4.6 [int], page 83; Section 5.3.11 [voice], page 303.

5.3.7 short

Type: int

Purpose: the output of monomials is done in the short manner, if `short` is non-zero. A C-like notion is used, if short is zero. Both notations may be used as input.
The default depends on the names of the ring variables (0 if there are names of variables longer than 1 character, 1 otherwise). Every change of the basering sets `short` to the previous value for that ring. In other words, the value of the variable `short` is "ring-local".

Example:

```
  ring r=23,x,dp;
  int save=short;
  short=1;
  2x2,x2;
↦ 2x2 x2
  short=0;
  2x2,x2;
↦ 2*x^2 x^2
  short=save;  //resets short to the previous value
```

See Section 4.6 [int], page 83.

5.3.8 timer

Type: int

Purpose:

1. the CPU time (i.e, user and system time) used for each command is printed if timer >0 , if this time is bigger than a (customizable) minimal time and if `printlevel+1 >= voice` (which is by default true on the Singular top level, but not true while procedures are executed).
2. yields the CPU time used since the start-up of Singular in a (customizable) resolution.

The default setting of `timer` is 0, the default minimal time is 0.5 seconds, and the default timer resolution is 1 (i.e., the default unit of time is one second). The minimal time and timer resolution can be set using the command line options `--min-time` and `--ticks-per-sec` and can be checked using `system("--min-time")` and `system("--ticks-per-sec")`.

How to use `timer` in order to measure the time for a sequence of commands, see example below.

Note for Windows95/98:
The value of the `timer` cannot be used (resp. trusted) when Singular is run under Windows95/98 (this is due to the shortcomings of the Windows95/98 operating system). Use Section 5.3.10 [rtimer], page 302, instead.

Example:

```
  timer=1; // The time of each command is printed
  int t=timer; // initialize t by timer
  ring r=0,(x,y,z),dp;
  poly p=(x+2y+3z+4xy+5xz+6yz)^20;
  // timer as int_expression:
  t=timer-t;
  t;  // yields the time in ticks-per-sec (default 1)
↦ 0
      // since t was initialized by timer
  int tps=system("--ticks-per-sec");
  t div tps; // yields the time in seconds truncated to int
↦ 0
  timer=0;
  system("--ticks-per-sec",1000); // set timer resolution to ms
  t=timer; // initialize t by timer
  p=(x+2y+3z+4xy+5xz+6yz)^20;
```

```
        timer-t;  // time in ms
      ↦ 30
```

See Section 3.1.6 [Command line options], page 19; Section 5.3.6 [printlevel], page 299; Section 5.3.10 [rtimer], page 302; Section 5.1.151 [system], page 270; Section 5.3.11 [voice], page 303.

5.3.9 TRACE

Type: int

Purpose: sets level of debugging.

`TRACE=0` No debugging messages are printed.

`TRACE=1` Messages about entering and leaving of procedures are displayed.

`TRACE=3` Messages about entering and leaving of procedures together with line numbers are displayed.

`TRACE=4` Each line is echoed and the interpretation of commands in this line is suspended until the user presses `RETURN`.

`TRACE=8` (debug version only:) show basering for all levels

`TRACE=128`
show all calls to kernel routines

`TRACE=256`
show all assigns

`TRACE=512`
show all automtic type conversions

`TRACE=1024`
profiling: print line numbers to smon.out

`TRACE` is defaulted to 0.
`TRACE` does not affect the output of commands.

Example:

```
        TRACE=1;
        LIB "general.lib";
        sum(1..100);
      ↦ entering sum (level 0)
      ↦ entering   lsum (level 1)
      ↦ entering     lsum (level 2)
      ↦ entering       lsum (level 3)
      ↦ entering         lsum (level 4)
      ↦ leaving          lsum (level 4)
      ↦ entering         lsum (level 4)
      ↦ leaving          lsum (level 4)
      ↦ leaving        lsum (level 3)
      ↦ entering       lsum (level 3)
      ↦ entering         lsum (level 4)
      ↦ leaving          lsum (level 4)
      ↦ entering         lsum (level 4)
      ↦ leaving          lsum (level 4)
```

```
↦ leaving        lsum (level 3)
↦ leaving      lsum (level 2)
↦ entering     lsum (level 2)
↦ entering       lsum (level 3)
↦ entering         lsum (level 4)
↦ leaving          lsum (level 4)
↦ entering         lsum (level 4)
↦ leaving          lsum (level 4)
↦ leaving        lsum (level 3)
↦ entering       lsum (level 3)
↦ entering         lsum (level 4)
↦ leaving          lsum (level 4)
↦ entering         lsum (level 4)
↦ leaving          lsum (level 4)
↦ leaving        lsum (level 3)
↦ leaving      lsum (level 2)
↦ leaving    lsum (level 1)
↦ entering   lsum (level 1)
↦ entering     lsum (level 2)
↦ entering       lsum (level 3)
↦ entering         lsum (level 4)
↦ leaving          lsum (level 4)
↦ entering         lsum (level 4)
↦ leaving          lsum (level 4)
↦ leaving        lsum (level 3)
↦ entering       lsum (level 3)
↦ entering         lsum (level 4)
↦ leaving          lsum (level 4)
↦ entering         lsum (level 4)
↦ leaving          lsum (level 4)
↦ leaving        lsum (level 3)
↦ leaving      lsum (level 2)
↦ entering     lsum (level 2)
↦ entering       lsum (level 3)
↦ entering         lsum (level 4)
↦ leaving          lsum (level 4)
↦ entering         lsum (level 4)
↦ leaving          lsum (level 4)
↦ leaving        lsum (level 3)
↦ entering       lsum (level 3)
↦ entering         lsum (level 4)
↦ leaving          lsum (level 4)
↦ entering         lsum (level 4)
↦ leaving          lsum (level 4)
↦ leaving        lsum (level 3)
↦ leaving      lsum (level 2)
↦ leaving    lsum (level 1)
↦ leaving  sum (level 0)
↦ 5050
```

See Section 4.6 [int], page 83.

5.3.10 rtimer

Type: int

Purpose: identical to `timer` (see Section 5.3.8 [timer], page 300), except that real times (i.e., wall-clock) times are reported, instead of CPU times. This can be trusted on all operating systems (including Windows95/98).

5.3.11 voice

Type: int

Purpose: shows the nesting level of procedures.

Note: See Section 3.8 [Libraries], page 55, for a small example how this is used for the display of comments while procedures are executed.

Example:

```
  voice;
↦ 1
proc p
{
  voice;
};
p();
↦ 2
```

See Section 5.1.16 [dbprint], page 165; Section 5.1.80 [listvar], page 210; Section 5.3.6 [printlevel], page 299.

6 Tricks and pitfalls

6.1 Limitations

SINGULAR has the following limitations:

- the characteristic of a prime field must be less than or equal to 2147483647 (2^31)
  (the characteristic of a prime field in the factory routines must be less than 536870912 (2^29))
  (the characteristic of a prime field in the NTL routines must be less than NTL_SP_BOUND (2^30) on 32bit machines - This is always the case since currently, only factory uses NTL.)
- the number of elements in GF(p,n) must be less than 65536
- the (weighted) degree of a monomial must be less or equal than 2147483647
- the rank of any free module must be less or equal than 2147483647
- the maximal allowed exponent of a ring variable depends on the ordering of the ring and is at least 32767. See also Section B.2 [Monomial orderings], page 696 for setting other limits.
- the precision of long floating point numbers (for ground field `real`) must be less or equal than 32767
- integers (of type `int`) have the limited range from -2147483648 to 2147483647
- floating point numbers (type `number` from field `real`) have a limited range which is machine dependent. A typical range is -1.0e-38 to 1.0e+38. The string representation of overflow and underflow is machine dependent, as well. For example `"Inf"` on Linux, or `"+.+00e+00"` on HPUX.
  Their input syntax is given by `scanf`, but must start with a digit.
- floating point numbers (type `number` from field `real` with a precision p larger then 3) use internally `mpf_set_default_prec(3.5*p+1)`.
  Their input syntax is given by `mpf_set_str` from GMP, but must start with a digit.
- the length of an identifier is unlimited but `listvar` displays only the first 20 characters
- statements may not contain more than 10000 tokens
- tokens (i.e. strings, numbers, ...) may not be longer than 16382 characters
- All input to SINGULAR must be 7-bit clean, i.e. special characters like the the German Umlaute (ä, ö, etc.), or the French accent characters may neither appear as input to SINGULAR, nor in libraries or procedure definitions.

6.2 System dependent limitations

Ports of SINGULAR to different systems do not always implement all possible parts of SINGULAR:

- dynamic modules are implemented for
  - unix systems with ELF format for executables (Linux, Solaris, FreeBSD)

6.3 Major differences to the C programming language

Although many constructs from SINGULAR's programming language are similar to those from the C programming language, there are some subtle differences. Most notably:

6.3.1 No rvalue of increments and assignments

The increment operator `++` (resp. decrement operator `--`) has no rvalue, i.e., cannot be used on the right-hand sides of assignments. So, instead of

```
j = i++;  // WRONG!!!
```

(which results in an error), it must be written

```
i++; j = i;
```

Likewise, an assignment expression does not have a result. Therefore, compound assignments like `i = j = k;` are not allowed and result in an error.

6.3.2 Evaluation of logical expressions

All arguments of a logical expression are first evaluated and then the value of the logical expression is determined. For example, the logical expressions `(a || b)` is evaluated by first evaluating `a` *and* `b`, even though the value of `b` has no influence on the value of `(a || b)`, if `a` evaluates to true.

Note, that this evaluation is different from the left-to-right, conditional evaluation of logical expressions (as found in most programming languages). For example, in these other languages, the value of `(1 || b)` is determined without ever evaluating `b`. This causes some problems with boolean tests on variables, which might not be defined at evaluation time. For example, the following results in an error, if the variable `i` is undefined:

```
if (defined(i) && i > 0) {} // WRONG!!!
```

This must be written instead as:

```
if (defined(i))
{
  if (i > 0) {}
}
```

However, there are several short work-arounds for this problem:

1. If a variable (say, `i`) is only to be used as a boolean flag, then define (value is TRUE) and undefine (value is FALSE) `i` instead of assigning a value. Using this scheme, it is sufficient to simply write

   ```
   if (defined(i))
   ```

   in order to check whether `i` is TRUE. Use the command `kill` to undefine a variable, i.e. to assign it a FALSE value (see Section 5.1.69 [kill], page 204).
2. If a variable can have more than two values, then define it, if necessary, before it is used for the first time. For example, if the following is used within a procedure

   ```
   if (! defined(DEBUG)) { int DEBUG = 1;}
   ...
   if (DEBUG == 3)  {...}
   if (DEBUG == 2)  {...}
   ...
   ```

   then a user of this procedure does not need to care about the existence of the `DEBUG` variable – this remains hidden from the user. However, if `DEBUG` exists globally, then its local default value is overwritten by its global one.

6.3.3 No case or switch statement

SINGULAR does not offer a `case` (or `switch`) statement. However, it can be imitated in the following way:

```
while (1)
{
   if (choice == choice_1) { ...; break;}
   ...
   if (choice == choice_n) { ...; break;}
   // default case
   ...; break;
}
```

6.3.4 Usage of commas

In SINGULAR, a comma separates list elements and the value of a comma expression is a list. Hence, commas cannot be used to combine several expressions into a single expression. For example, instead of writing

```
for (i=1, j=5; i<5 || j<10; i++, j++) {...} // WRONG!!!!!!
```

one has to write

```
for (i,j = 1,5; i<5 || j<10; i++, j++) {...}
```

6.3.5 Usage of brackets

In SINGULAR, curly brackets (`{ }`) **must always** be used to enclose the statement body following such constructs like `if`, `else`, `for`, or `while`, even if this block consists of only a single statement. Similarly, in the return statement of a procedure, parentheses (`( )`) **must always** be used to enclose the return value. Even if there is no value to return, parentheses have to be used after a return statement (i.e., `return();`). For example,

```
if (i == 1) return i;    // WRONG!!!!!
```

results in an error. Instead, it must be written as

```
if (i == 1) { return (i); }.
```

6.3.6 Behavior of continue

SINGULAR's `continue` construct is only valid inside the body of a `for` or `while` construct. It skips the rest of the loop-body and jumps to the beginning of the block. Unlike the C-construct SINGULAR's `continue` **does not execute the increment statement**. For example,

```
for (int i = 1 ; i<=10; i=i+1)
{
   ...
   if (i==3) { i=8;continue; }
     // skip the rest if i is 3 and
     // continue with the next i: 8
   i;
}
↦ 1
↦ 2
↦ 8
↦ 9
↦ 10
```

6.3.7 Return type of procedures

Although the SINGULAR language is a strongly typed programming language, the type of the return value of a procedure does not need to be specified. As a consequence, the return type of a procedure may vary, i.e., may, for example, depend on the input. However, the return value of such a procedure may then only be assigned to a variable of type `def`.

```
proc type_return (int i)
{
  if (i > 0) {return (i);}
  else {return (list(i));}
}
def t1 = type_return(1);
def t2 = type_return(-1);
typeof(t1); typeof(t2);
↦ int
↦ list
```

Furthermore, it is mandatory to assign the return value of a procedure to a variable of type `def`, if a procedure changes the current ring using the `keepring` command (see Section 5.2.11 [keepring], page 293) and returns a ring-dependent value (like a polynomial or module).

```
proc def_return
{
  ring r=0,(x,y),dp;
  poly p = x;
  keepring r;
  return (x);
}
def p = def_return();
// poly p = def_return(); would be WRONG!!!
typeof(p);
↦ poly
```

On the other hand, more than one value can be returned by a single `return` statement. For example,

```
proc tworeturn () { return (1,2); }
int i,j = tworeturn();
```

6.3.8 First index is 1

Although the SINGULAR language is C like, the indices of all objects which may have an index start at 1.

```
  ring r;
  ideal i=1,x,z;
  i[2];
↦ x
  intvec v=1,2,3;
  v[1];
↦ 1
  poly p=x+y+z;
  p[2];
↦ y
  vector h=[x+y,x,z];
  h[1];
```

```
↦ x+y
  h[1][1];
↦ x
```

6.4 Miscellaneous oddities

1. integer division

   If two numerical constants (i.e., two sequences of digits) are divided using the `/` operator, the surrounding whitespace determines which division to use: if there is no space between the constants and the `/` operator (e.g., "3/2"), both numerical constants are treated as of type `number` and the current ring division is used. If there is at least one space surrounding the `/` operator (e.g., "3 / 2"), both numerical constants are treated as of type `int` and an integer division is performed. To avoid confusion, use the `div` operator instead of `/` for integer division and an explicit type cast to `number` for ring division. Note, that this problem does only occur for divisions of numerical constants. It also apllies for large numerical constants which are of type `bigint`.

   ```
     ring r=32002,x,dp;
   ↦ // ** 32002 is invalid as characteristic of the ground field. 32003 is us\
      ed.
     3/2;    // ring division
   ↦ -16000
     3 / 2;  // integer division
   ↦ // ** int division with ‘/‘: use ‘div‘ instead in line >>  3 / 2;  // int\
      eger division<<
   ↦ 1
     3 div 2;
   ↦ 1
     number(3) / number(2);
   ↦ -16000
     number a=3;
     number b=2;
     a/b;
   ↦ -16000
     int c=3;
     int d=2;
     c / d;
   ↦ // ** int division with ‘/‘: use ‘div‘ instead in line >>  c / d;<<
   ↦ 1
   ```

2. monomials and precedence

   The formation of a monomial has precedence over all operators (a monomial is here an optional coefficient followed by any sequence of ring variables (possibly followed by an exponent) which only conssist of letters, digits and (over the rationals) `/` without any whitespace):

   ```
     ring r=0,(x,y),dp;
     2xy^2 == (2*x*y)^2;
   ↦ 1
     2xy^2 == 2x*y^2;
   ↦ 0
     2x*y^2 == 2*x * (y^2);
   ↦ 1
   ```

   During that formation no operator is involved: in the non-commutative case, we have

```
  LIB "nctools.lib";
  ring r = 0,(x,y),dp;
  def S = superCommutative();
  xy == yx;
↦ 1
  x*y == y*x;
↦ 1
  x*y, y*x;
↦ xy xy
```

3. meaning of `mult`

   For an arbitrary ideal or module `i`, `mult(i)` returns the multiplicity of the ideal generated by the leading monomials of the given generators of `i`, hence depends on the monomial ordering!

   A standard mistake is to interpret `degree(i)` or `mult(i)` for an inhomogeneous ideal `i` as the degree of the homogenization or as something like the 'degree of the affine part'. For the ordering `dp` (degree reverse lexicographical) the converse is true: if `i` is given by a standard basis, `mult(i)` is the degree of the homogeneous ideal obtained by homogenization of `i` and then putting the homogenizing variable to 0, hence it is the degree of the part at infinity (this can also be checked by looking at the initial ideal).

4. size of ideals

   `size` counts the non-zero entries of an ideal or module. Use `ncols` to determine the actual number of entries in the ideal or module.

5. computations in `qring`

   In order to speed up computations in quotient rings, Singular usually does not reduce polynomials w.r.t. the quotient ideal; rather the given representative is used as long as possible during computations. If it is necessary, reduction is done during standard base computations. To reduce a polynomial `f` by hand w.r.t. the current quotient ideal use the command `reduce(f,std(0))` (see Section 5.1.127 [reduce], page 246).

6. degree of a polynomial

   `degBound`
   The exact meaning of "degree" depends on the ring odering and the command: `slimgb` uses always the total degree with weights 1, `std` does so only for block orderings.

   `hilb`
   the degree is the total degree with weights 1 unless a weight vector is given

   `kbase`
   the degree is the total degree with weights 1 (to use another weight vector see [weightKB], page 721)

7. substring selection

   To extract substrings from a `string`, square brackets are used, enclosing either two comma-separated `int`s or an `intvec`. Although two comma-separated `int`s represent an `intvec`, they mean different things in substring access. Square brackets enclosing two `int`s (e.g. `s[2,6]`) return a substring where the first integer denotes the starting position and the second integer denotes the length of the substring. The result is returned as a `string`. Square brackets enclosing an `intvec` (e.g. `s[intvec(2,6)]`) return the characters of the string at the position given by the values of the `intvec`. The result is returned as an expression list of strings.

```
  string s = "one-word";
  s[2,6];     // a substring starting at the second char
↦ ne-wor
  size(_);
```

```
↦ 6
  intvec v = 2,6;
  s[v];      // the second and the sixth char
↦ n o
  string st = s[v];  // stick together by an assignment
  st;
↦ no
  size(_);
↦ 2
  v = 2,6,8;
  s[v];
↦ n o d
```

8. packages and indexed variables

   See example

```
package K;
string K::varok;
string K::donotwork(1);
int K::i(1..3);
// Toplevel does not contain i(1..3)
listvar();
↦ // i(3)                          [0]  int 0
↦ // i(2)                          [0]  int 0
↦ // i(1)                          [0]  int 0
↦ // donotwork(1)                  [0]  string
// i(1..3) are stored in Package 'K'
listvar(K);
↦ // K                             [0]  package K (N)
↦ // ::varok                       [0]  string
```

6.5 Identifier resolution

In Singular, an identifier (i.e., a "word") is resolved in the following way and order: It is checked for

1. a reserved name (like `ring`, `std`, ...),
2. a local variable (w.r.t. a procedure),
3. a local ring variable (w.r.t. the current basering locally set in a procedure),
4. a global variable,
5. a global ring variable (w.r.t. the current basering)
6. a monomial consisting of local ring variables written without operators,
7. a monomial consisting of global ring variables written without operators.

Consequently, it is allowed to have general variables with the same name as ring variables. However, the above identifier resolution order must be kept in mind. Otherwise, surprising results may come up.

```
ring r=0,(x,y),dp;
int x;
x*y; // resolved product int*poly, i.e., 0*y
↦ 0
xy; // "xy" is one identifier and resolved to monomial xy
↦ xy
```

For these reasons, we strongly recommend not to use variables which have the same name(s) as ring variables.

Moroever, we strongly recommend not to use ring variables whose name is fully contained in (i.e., is a substring of) another name of a ring variable. Otherwise, effects like the following might occur:

```
ring r=0,(x, x1),dp; // name x is substring of name x1 !!!!!!!!!
x;x1;   // resolved polynomial x
↦ x
↦ x1
short=0; 2x1; // resolved to monomial 2*x^1 !!!!!!
↦ 2*x
2*x1; // resolved to product 2 times x1
↦ 2*x1
```

7 Non-commutative subsystem

7.1 PLURAL

What is and what does PLURAL**?**

PLURAL is a kernel extension of SINGULAR, providing many algorithms for computations within certain non-commutative algebras (see Section 7.4 [Mathematical background (plural)], page 359 for detailed information on algebras and algorithms).

It uses the same data structures, sometimes interpreting them in a different way and/or modifying them for its own purposes. In spite of such a difference, one can always transfer objects between commutative rings of SINGULAR and non-commutative rings.

With PLURAL, one can set up a non-commutative G -algebra, say A , with a Poincaré-Birkhoff-Witt (PBW) basis, (see Section 7.4.1 [G-algebras], page 359 for step-by-step building instructions and also Section 7.7 [Non-commutative libraries], page 370 for procedures for setting many important algebras easily).

Functionalities of PLURAL (enlisted in Section 7.3 [Functions (plural)], page 329) are accessible as soon as the basering becomes non-commutative (see Section 7.3.16 [nc_algebra], page 342).

One can perform various computations with polynomials and ideals in A and with vectors and submodules of a free module A^n.

One can work also within factor algebras of G -algebras (see Section 7.2.5 [qring (plural)], page 324 type) by two-sided ideals (see Section 7.3.29 [twostd], page 358).

What PLURAL **does not:**

PLURAL does not perform computations in the free algebra or in its general factor algebras (instead, these computations can be possibly done due to Section 7.6 [LETTERPLACE], page 367).

In PLURAL one can only work with G -algebras and with their factor-algebras by two-sided ideals (GR -algebras).

PLURAL requires a monomial ordering but it does not work generally with local and mixed orderings. Right now, one can use only global orderings in PLURAL (see Section B.2.2 [General definitions for orderings], page 696), save for SCA, where we provide the possibility of computations in a tensor product of a non-commutative algebra (with a global ordering) with a commutative algebra (with any ordering). In the future, this will be enhanced for other algebras, as well.

PLURAL does not handle non-commutative parameters.

Defining parameters, one **cannot** impose non-commutative relations on them. Moreover, it is impossible to introduce parameters which do not commute with variables.

PLURAL **conventions**

***-multiplication (plural)**

in the non-commutative case, the correct multiplication of `y` by `x` must be written as `y*x`.
Both expressions `yx` and `xy` are equal, since they are interpreted as commutative expressions. See example in Section 7.2.4.2 [poly expressions (plural)], page 323.
Note, that PLURAL output consists only of monomials, hence the signs `*` are omitted.

`ideal` **(plural)**
: Unless stated otherwise, an `ideal` as understood by PLURAL, is a list of generators of a **left** ideal. For more information see Section 7.2.1 [ideal (plural)], page 313.
For a **two-sided ideal** `T`, use command Section 7.3.29 [twostd], page 358 in order to compute the two-sided Groebner basis of `T`.

`module` **(plural)**
: Unless stated otherwise, a `module` as understood by PLURAL, is **either** a fininitely generated **left** submodule of a free module (of finite rank)
or a factor module of a free module (of finite rank) by its left submodule (see Section 7.2.3 [module (plural)], page 319 for details).

`qring` **(plural)**
: In PLURAL it is only possible to build factor-algebras modulo **two-sided** ideals (see Section 7.2.5 [qring (plural)], page 324).

7.2 Data types (plural)

This chapter explains all data types of PLURAL in alphabetical order. For every type, there is a description of the declaration syntax
as well as information about how to build expressions of certain types.

The term "expression list" in PLURAL refers to any comma separated list of expressions.

For the general syntax of a declaration see Section 3.5.1 [General command syntax], page 41.

7.2.1 ideal (plural)

For PLURAL ideals are **left** ideals, unless stated otherwise.
Ideals are represented as lists of polynomials which are interpreted as left generators of the ideal.
For the operations with two-sided ideals see Section 7.3.29 [twostd], page 358.

Like polynomials, ideals can only be defined or accessed with respect to a basering.

Note: `size` counts only the non-zero generators of an ideal whereas `ncols` counts all generators.

7.2.1.1 ideal declarations (plural)

Syntax: `ideal` name = list_of_poly_and_ideal_expressions `;`
`ideal` name = ideal_expression `;`

Purpose: defines a left ideal.

Default: 0

Example:

```
ring r=0,(x,y,z),dp;
def R=nc_algebra(-1,0); // an anti-commutative algebra
setring R;
poly s1 = x2;
poly s2 = y3;
poly s3 = z;
ideal i =  s1, s2-s1, 0,s3*s2, s3^4;
i;
↦ i[1]=x2
↦ i[2]=y3-x2
↦ i[3]=0
```

```
↦ i[4]=-y3z
↦ i[5]=z4
size(i);
↦ 4
ncols(i);
↦ 5
```

7.2.1.2 ideal expressions (plural)

An ideal expression is:

1. an identifier of type ideal
2. a function returning an ideal
3. a combination of ideal expressions by the arithmetic operations `+` or `*`
4. a power of an ideal expression (operator `^` or `**`)
   Note that the computation of the product `i*i` involves all products of generators of `i` while `i^2` involves only the different ones, and is therefore faster.
5. a type cast to ideal

Example:

```
ring r=0,(x,y,z),dp;
def R=nc_algebra(-1,0); // an anticommutative algebra
setring R;
ideal m = maxideal(1);
m;
↦ m[1]=x
↦ m[2]=y
↦ m[3]=z
poly f = x2;
poly g = y3;
ideal i = x*y*z , f-g, g*(x-y) + f^4 ,0, 2x-z2y;
ideal M = i + maxideal(10);
i = M*M;
ncols(i);
↦ 598
i = M^2;
ncols(i);
↦ 690
i[ncols(i)];
↦ x20
vector v = [x,y-z,x2,y-x,x2yz2-y];
ideal j = ideal(v);
j;
↦ j[1]=x
↦ j[2]=y-z
↦ j[3]=x2
↦ j[4]=-x+y
↦ j[5]=x2yz2-y
```

7.2.1.3 ideal operations (plural)

`+` addition (concatenation of the generators and simplification)

`*` multiplication (with ideal, poly, vector, module; in case of multiplication with ideal or module, the result will be simplified)

`^` exponentiation (by a non-negative integer)

ideal_expression `[` intvec_expression `]`
are polynomial generators of the ideal, index 1 gives the first generator.

Note: For simplification of an ideal, see also Section 5.1.139 [simplify], page 258.

Example:

```
ring r=0,(x,y,z),dp;
matrix D[3][3];
D[1,2]=-z;  D[1,3]=y;  D[2,3]=x;
def R=nc_algebra(1,D); // this algebra is U(so_3)
setring R;
ideal I = 0,x,0,1;
I;
↦ I[1]=0
↦ I[2]=x
↦ I[3]=0
↦ I[4]=1
I + 0;    // simplification
↦ _[1]=1
I*x;
↦ _[1]=0
↦ _[2]=x2
↦ _[3]=0
↦ _[4]=x
ideal J = I,0,x,x-z;
I * J;   //  multiplication with simplification
↦ _[1]=1
vector V = [x,y,z];
print(I*V);
↦ 0,x2,0,x,
↦ 0,xy,0,y,
↦ 0,xz,0,z
ideal m = maxideal(1);
m^2;
↦ _[1]=x2
↦ _[2]=xy
↦ _[3]=xz
↦ _[4]=y2
↦ _[5]=yz
↦ _[6]=z2
ideal II = I[2..4];
II;
↦ II[1]=x
↦ II[2]=0
↦ II[3]=1
```

7.2.1.4 ideal related functions (plural)

`dim` Gelfand-Kirillov dimension of basering modulo the ideal of leading terms (see Section 7.3.3 [dim (plural)], page 331)

`eliminate` elimination of variables (see Section 7.3.5 [eliminate (plural)], page 333)

`intersect` ideal intersection (see Section 7.3.9 [intersect (plural)], page 336)

`kbase` vector space basis of basering modulo the leading ideal (see Section 7.3.10 [kbase (plural)], page 337)

`lead` leading terms of a set of generators (see Section 5.1.73 [lead], page 206)

`lift` lift-matrix (see Section 7.3.11 [lift (plural)], page 338)

`liftstd` left Groebner basis and transformation matrix computation (see Section 7.3.12 [liftstd (plural)], page 339)

`maxideal` generators of a power of the maximal ideal at 0 (see Section 5.1.86 [maxideal], page 216)

`modulo` represents $(h1+h2)/h1 \cong h2/(h1\cap h2)$ (see Section 7.3.14 [modulo (plural)], page 341)

`mres` minimal free resolution of an ideal and a minimal set of generators of the given ideal (see Section 7.3.15 [mres (plural)], page 341)

`ncols` number of columns (see Section 5.1.101 [ncols], page 227)

`nres` computes a free resolution of an ideal resp. module M which is minimized from the second free module on (see Section 7.3.18 [nres (plural)], page 345)

`oppose` creates an opposite ideal of a given ideal from the given ring into a basering (see Section 7.3.19 [oppose], page 346)

`preimage` preimage under a ring map (see Section 7.3.21 [preimage (plural)], page 348)

`quotient` ideal quotient (see Section 7.3.22 [quotient (plural)], page 349)

`reduce` left normal form with respect to a left Groebner basis (see Section 7.3.23 [reduce (plural)], page 350)

`simplify` simplify a set of polynomials (see Section 5.1.139 [simplify], page 258)

`size` number of non-zero generators (see Section 5.1.140 [size], page 260)

`slimgb` left Groebner basis computation with slim technique (see Section 7.3.25 [slimgb (plural)], page 353)

`std` left Groebner basis computation (see Section 7.3.26 [std (plural)], page 355)

`subst` substitute a ring variable (see Section 7.3.27 [subst (plural)], page 356)

`syz` computation of the first syzygy module (see Section 7.3.28 [syz (plural)], page 357)

`twostd` two-sided Groebner basis computation (see Section 7.3.29 [twostd], page 358)

`vdim` vector space dimension of basering modulo the leading ideal (see Section 7.3.30 [vdim (plural)], page 358)

7.2.2 map (plural)

Maps are ring maps from a preimage ring (source) into the basering (target), defined by specifying images for source variables in the target ring.

Note:

- the target of a map is **ALWAYS** the actual basering
- the preimage ring has to be stored "by its name", that means, maps can only be used in such contexts, where the name of the preimage ring can be resolved (this has to be considered in subprocedures). See also Section 6.5 [Identifier resolution], page 310, Section 3.7.4 [Names in procedures], page 55.

Maps between rings with different coefficient fields are possible and listed below.

Canonically realized are

- $Q \to Q(a, \ldots)$ (Q : the rational numbers)
- $Q \to R$ (R : the real numbers)
- $Q \to C$ (C : the complex numbers)
- $Z/p \to (Z/p)(a, \ldots)$ (Z : the integers)
- $Z/p \to GF(p^n)$ (GF : the Galois field)
- $Z/p \to R$
- $R \to C$

Possible are furthermore

- $Z/p \to Q, \quad [i]_p \mapsto i \in [-p/2,\, p/2] \subseteq Z$
- $Z/p \to Z/p', \quad [i]_p \mapsto i \in [-p/2,\, p/2] \subseteq Z,\ i \mapsto [i]_{p'} \in Z/p'$
- $C \to R$, by taking the real part

Finally, in PLURAL we allow the mapping from rings with coefficient field Q to rings whose ground fields have finite characteristic:

- $Q \to Z/p$
- $Q \to (Z/p)(a, \ldots)$

Note: In these cases the denominator and the numerator of a number are mapped separately by the usual map from Z to Z/p, and the image of the number is built again afterwards by division. It is thus not allowed to map numbers whose denominator is divisible by the characteristic of the target ground field, or objects containing such numbers. We, therefore, strongly recommend to use such maps only to map objects with integer coefficients.
Note that - in contrast to the commutative case - maps between non-commutative rings easily fail to be a morphism.

7.2.2.1 map declarations (plural)

Syntax: `map` name = preimage_ring_name , ideal_expression ;
`map` name = preimage_ring_name , list_of_poly_and_ideal_expressions ;
`map` name = map_expression ;

Purpose: defines a ring map from `preimage_ring` to basering.
Maps the variables of the `preimage ring` to the generators of the ideal.
If the ideal contains less elements than the number of variables in the `preimage_ring`, the remaining variables are mapped to 0.
If the ideal contains more elements, extra elements are ignored.

The image ring is always the current basering. For the mapping of coefficients from different fields see Section 7.2.2 [map (plural)], page 317.

Default: none

Note: There are standard mappings for maps which are close to the identity map: `fetch (plural)` and `imap (plural)`.

The name of a map serves as the function which maps objects from the preimage_ring into the basering. These objects must be defined by names (no evaluation in the preimage ring is possible).

Example:

```
// an easy example
ring r1 = 0,(a,b),dp; // a commutative ring
poly P = a^2+ab+b^3;
ring r2 = 0,(x,y),dp;
def W=nc_algebra(1,-1); // a Weyl algebra
setring W;
map M = r1, x^2, -y^3;
// note: M is a map and not a morphism
M(P);
↦ -y9-x2y3+x4
// now, a more involved example
LIB "ncalg.lib";
def Usl2 = makeUsl2();
// this algebra is U(sl_2), generated by  e,f,h
setring Usl2;
poly  P  = 4*e*f+h^2-2*h; // the central el-t of Usl2
poly  Q  = e^3*f-h^4;     // some polynomial
ring W1  = 0,(D,X),dp;
def W2=nc_algebra(1,-1);
setring W2;
// this algebra is the opposite Weyl algebra
map F = Usl2, -X, D*D*X, 2*D*X;
F(P); // 0, because P is in the kernel of F
↦ 0
F(Q);
↦ -16D4X4+96D3X3-D2X4-112D2X2+6DX3+16DX-6X2
```

See Section 7.3.7 [fetch (plural)], page 335; Section 7.2.1.2 [ideal expressions (plural)], page 314; Section 7.3.8 [imap (plural)], page 335; Section 7.2.2 [map (plural)], page 317; Section 7.2.7 [ring (plural)], page 327.

7.2.2.2 map expressions (plural)

A map expression is:

1. an identifier of type map
2. a function returning map
3. a composition of maps using parentheses, e.g. `f(g)`

7.2.2.3 map (plural) operations

`( )` composition of maps. If, for example, `f` and `g` are maps, then `f(g)` is a map expression giving the composition $f \circ g$ of `f` and `g`, provided the target ring of `g` is the basering of `f`.

map_expression `[` int_expressions `]`
is a map entry (the image of the corresponding variable)

Example:

```
LIB "ncalg.lib";
def Usl2 = makeUsl2(); // this algebra is U(sl_2)
setring Usl2;
map F = Usl2, f, e, -h; // involutive endomorphism of U(sl_2)
F;
↦ F[1]=f
↦ F[2]=e
↦ F[3]=-h
map G = F(F);
G;
↦ G[1]=e
↦ G[2]=f
↦ G[3]=h
poly p = (f+e*h)^2 + 3*h-e;
p;
↦ e2h2+2e2h+2efh-2ef+f2-h2-e+3h
F(p);
↦ f2h2-2efh-2f2h+e2-2ef+h2-f-h
G(p);
↦ e2h2+2e2h+2efh-2ef+f2-h2-e+3h
(G(p) == p); // G is the identity
↦ 1
```

7.2.2.4 map related functions (plural)

`fetch (plural)`
the identity map between rings and qrings (see Section 7.3.7 [fetch (plural)], page 335)

`imap (plural)`
a convenient map procedure for inclusions and projections of rings (see Section 7.3.8 [imap (plural)], page 335)

`preimage (plural)`
preimage under a ring map (see Section 7.3.21 [preimage (plural)], page 348)

`subst` substitute a ring variable (see Section 7.3.27 [subst (plural)], page 356)

7.2.3 module (plural)

Modules are **left** submodules of a free module over the basering with basis `gen(1)`, `gen(2)`, ..., `gen(n)` for some natural number `n`.
They are represented by lists of vectors, which generate the left submodule. Like vectors, they can only be defined or accessed with respect to a basering.

If M is a left submodule of R^n (where R is the basering) generated by vectors $v_1, \ldots, v_k$, then these generators may be considered as the generators of relations of R^n/M between the canonical

generators `gen(1)`,...,`gen(n)`. Hence, any finitely generated R -module can be represented in PLURAL by its module of relations. This is the so-called Coker-representation.
The assignments `module M=v1,...,vk; matrix A=M;` create the presentation matrix of size $n \times k$, with the columns of A being the vectors $v_1, \ldots, v_k$ which generate M.

7.2.3.1 module declarations (plural)

Syntax: `module` name = list_of_vector_expressions (which are interpreted as left generators of the module) ;
`module` name = module_expression ;

Purpose: defines a left module.

Default: [0]

Example:

```
ring r=0,(x,y,z),(c,dp);
matrix D[3][3];
D[1,2]=-z;  D[1,3]=y;  D[2,3]=x;
def R=nc_algebra(1,D); // this algebra is U(so_3)
setring R;
vector s1 = [x2,y3,z];
vector s2 = [xy,1,0];
vector s3 = [0,x2-y2,z];
poly   f  = -x*y;
module m = s1, s2-s1,f*(s3-s1);
m;
↦ m[1]=[x2,y3,z]
↦ m[2]=[-x2+xy,-y3+1,-z]
↦ m[3]=[x3y-2x2z-xy,xy4-x3y+xy3+2x2z+xy]
// show m in matrix format (columns generate m)
print(m);
↦ x2,-x2+xy,x3y-2x2z-xy,
↦ y3,-y3+1, xy4-x3y+xy3+2x2z+xy,
↦ z, -z,    0
```

7.2.3.2 module expressions (plural)

A module expression is:

1. an identifier of type module
2. a function returning module
3. module expressions combined by the arithmetic operation `+`
4. multiplication of a module expression with an ideal or a poly expression: `*`
5. a type cast to module

7.2.3.3 module operations (plural)

`+` addition (concatenation of the generators and simplification) Note that "-" implicitely converts a module into a matrix; see below example.

`*` right or left multiplication with number, ideal, or poly (but not 'module' * 'module'!)

module_expression `[` int_expression `,` int_expression `]`
: is a module entry, where the first index indicates the row and the second the column

module_expressions `[` int_expression `]`
: is a vector, where the index indicates the column (generator)

Example:

```
ring A=0,(x,y,z),Dp;
matrix D[3][3];
D[1,2]=-z;  D[1,3]=y;  D[2,3]=x;  // this algebra is U(so_3)
def B=nc_algebra(1,D);
setring B;
module M = [x,y],[0,0,x*z];
module N = matrix((x+y-z)*M) - matrix(M*(x+y-z)); // no - for type module
print(N);
↦ -y-z,0,
↦ -x+z,0,
↦ 0,   -x2-xy-yz-z2
```

7.2.3.4 module related functions (plural)

`eliminate`
: elimination of variables (see Section 7.3.5 [eliminate (plural)], page 333)

`freemodule`
: the free module of given rank (see Section 5.1.45 [freemodule], page 185)

`intersect`
: module intersection (see Section 7.3.9 [intersect (plural)], page 336)

`kbase` vector space basis of free module over the basering modulo the module of leading terms (see Section 7.3.10 [kbase (plural)], page 337)

`lead` initial module (see Section 5.1.73 [lead], page 206)

`lift` lift-matrix (see Section 7.3.11 [lift (plural)], page 338)

`liftstd` left Groebner basis and transformation matrix computation (see Section 7.3.12 [liftstd (plural)], page 339)

`modulo` represents $(h1+h2)/h1 \cong h2/(h1 \cap h2)$ (see Section 7.3.14 [modulo (plural)], page 341)

`mres` minimal free resolution of a module and a minimal set of generators of the given ideal module (see Section 7.3.15 [mres (plural)], page 341)

`ncols` number of columns (see Section 5.1.101 [ncols], page 227)

`nres` computes a free resolution of an ideal resp. module M which is minimized from the second free module on (see Section 7.3.18 [nres (plural)], page 345)

`nrows` number of rows (see Section 5.1.104 [nrows], page 229)

`oppose` creates an opposite module of a given module from the given ring into a basering (see Section 7.3.19 [oppose], page 346)

`print` nice print format (see Section 5.1.117 [print], page 238)

`prune` minimize the embedding into a free module (see Section 5.1.119 [prune], page 241) `quotient` module quotient (see Section 7.3.22 [quotient (plural)], page 349)

`reduce` left normal form with respect to a left Groebner basis (see Section 7.3.23 [reduce (plural)], page 350)

`simplify` simplify a set of vectors (see Section 5.1.139 [simplify], page 258)

`size` number of non-zero generators (see Section 5.1.140 [size], page 260)

`std` left Groebner basis computation (see Section 7.3.26 [std (plural)], page 355)

`subst` substitute a ring variable (see Section 7.3.27 [subst (plural)], page 356)

`syz` computation of the first syzygy module (see Section 7.3.28 [syz (plural)], page 357)

`vdim` vector space dimension of free module over the basering modulo module of leading terms (see Section 7.3.30 [vdim (plural)], page 358)

7.2.4 poly (plural)

Polynomials and vectors are the basic data for all main algorithms in Plural. Polynomials consist of finitely many terms (coefficient*monomial) which are combined by the usual polynomial operations (see Section 7.2.4.2 [poly expressions (plural)], page 323). Polynomials can only be defined or accessed with respect to a basering which determines the coefficient type, the names of the indeterminants and the monomial ordering.

Example:

```
ring r=32003,(x,y,z),dp;
poly f=x3+y5+z2;
```

Remark: Remember the conventions on polynomial multiplication we follow (`*`-multiplication in Section 7.1 [PLURAL], page 312).

7.2.4.1 poly declarations (plural)

Syntax: `poly` name `=` poly_expression `;`

Purpose: defines a polynomial.

Default: 0

Example:

```
ring r = 32003,(x,y,z),dp;
def R=nc_algebra(-1,1);
setring R;
// ring of some differential-like operators
R;
↦ // coefficients: ZZ/32003
↦ // number of vars : 3
↦ //        block   1 : ordering dp
↦ //                  : names    x y z
↦ //        block   2 : ordering C
↦ // noncommutative relations:
↦ //    yx=-xy+1
↦ //    zx=-xz+1
↦ //    zy=-yz+1
yx;      // not correct input
↦ xy
```

```
y*x;     // correct input
↦ -xy+1
poly s1  = x3y2+151x5y+186xy6+169y9;
poly s2  = 1*x^2*y^2*z^2+3z8;
poly s3  = 5/4x4y2+4/5*x*y^5+2x2y2z3+y7+11x10;
int a,b,c,t=37,5,4,1;
poly f=3*x^a+x*y^(b+c)+t*x^a*y^b*z^c;
f;
↦ x37y5z4+3x37+xy9
short = 0;
f;
↦ x^37*y^5*z^4+3*x^37+x*y^9
```

7.2.4.2 poly expressions (plural)

A polynomial expression is (optional parts in square brackets):

1. a monomial (there are NO spaces allowed inside a monomial)

   `[coefficient] ring_variable [exponent] [ring_variable [exponent] ...]`

   monomials which contain an indexed ring variable must be built from `ring_variable` and `coefficient` with the operations `*` and `^`
2. an identifier of type poly
3. a function returning poly
4. polynomial expressions combined by the arithmetic operations `+`, `-`, `*`, `/`, or `^`.
5. a type cast to poly

Example:

```
ring r=0,(x,y),dp;
def R=nc_algebra(1,1);  // make it a Weyl algebra
setring R;
R;
↦ // coefficients: QQ
↦ // number of vars : 2
↦ //        block   1 : ordering dp
↦ //                  : names    x y
↦ //        block   2 : ordering C
↦ // noncommutative relations:
↦ //    yx=xy+1
yx;      // not correct input
↦ xy
y*x;     // correct input
↦ xy+1
poly f = 10x2*y3 + 2y2*x^2 - 2*x*y + y - x + 2;
lead(f);
↦ 10x2y3
leadmonom(f);
↦ x2y3
simplify(f,1);     // normalize leading coefficient
↦ x2y3+1/5x2y2+3/5xy-1/10x+1/10y+3/5
cleardenom(f);
↦ 10x2y3+2x2y2+6xy-x+y+6
```

7.2.4.3 poly operations (plural)

`+` addition

`-` negation or subtraction

`*` multiplication

`/` commutative division by a monomial, non divisible terms yield 0

`^`, `**` power by a positive integer

`<`, `<=`, `>`, `>=`, `==`, `<>`
comparison (of leading monomials w.r.t. monomial ordering)

poly_expression `[` intvec_expression `]`
the sum of monomials at the indicated places w.r.t. the monomial ordering

7.2.4.4 poly related functions (plural)

`bracket` computes the Lie bracket of two polinomials (see Section 7.3.2 [bracket], page 330)

`lead` leading term (see Section 5.1.73 [lead], page 206)

`leadcoef` coefficient of the leading term (see Section 5.1.74 [leadcoef], page 207)

`leadexp` the exponent vector of the leading monomial (see Section 5.1.75 [leadexp], page 207)

`leadmonom`
leading monomial (see Section 5.1.76 [leadmonom], page 208)

`oppose` creates an opposite polynomial of a given polynomial from the given ring into a basering (see Section 7.3.19 [oppose], page 346)

`reduce` left normal form with respect to a left Groebner basis (see Section 7.3.23 [reduce (plural)], page 350)

`simplify` normalize a polynomial (see Section 5.1.139 [simplify], page 258)

`size` number of monomials (see Section 5.1.140 [size], page 260)

`subst` substitute a ring variable (see Section 7.3.27 [subst (plural)], page 356)

`var` the indicated variable of the ring (see Section 5.1.160 [var], page 279)

7.2.5 qring (plural)

PLURAL offers the possibility to compute within factor-rings modulo two-sided ideals. The ideal has to be given as a two-sided Groebner basis (see Section 7.3.29 [twostd], page 358 command).

For a detailed description of the concept of rings and quotient rings see Section 3.3 [Rings and orderings], page 30.

Note: we highly recommend to turn on `option(redSB); option(redTail);` while computing in qrings. Otherwise results may have a difficult interpretation.

7.2.5.1 qring declaration (plural)

Syntax: `qring` name = ideal_expression ;

Default: none

Purpose: declares a quotient ring as the basering modulo an `ideal_expression` and sets it as current basering.

Note: reports error if an ideal is not a two-sided Groebner basis.

Example:

```
ring r=0,(z,u,v,w),dp;
def R=nc_algebra(-1,0); // an anticommutative algebra
setring R;
option(redSB);
option(redTail);
ideal i=z^2,u^2,v^2,w^2, zuv-w;
qring Q = i;  // incorrect call produces error
↦ // ** i is no standard basis
↦ // ** i is no twosided standard basis
kill Q;
setring R;  // go back to the ring R
qring q=twostd(i); // now it is an exterior algebra modulo <zuv-w>
q;
↦ // coefficients: QQ
↦ // number of vars : 4
↦ //        block   1 : ordering dp
↦ //                  : names    z u v w
↦ //        block   2 : ordering C
↦ // noncommutative relations:
↦ //    uz=-zu
↦ //    vz=-zv
↦ //    wz=-zw
↦ //    vu=-uv
↦ //    wu=-uw
↦ //    wv=-vw
↦ // quotient ring from ideal
↦ _[1]=w2
↦ _[2]=vw
↦ _[3]=uw
↦ _[4]=zw
↦ _[5]=v2
↦ _[6]=u2
↦ _[7]=z2
↦ _[8]=zuv-w
poly k = (v-u)*(zv+u-w);
k; // the output is not yet totally reduced
↦ zuv-uv+uw-vw
poly ek=reduce(k,std(0));
ek; // the reduced form
↦ -uv+w
```

7.2.5.2 qring related functions (plural)

`envelope` enveloping ring (see Section 7.3.6 [envelope], page 334)

`nvars` number of ring variables (see Section 5.1.106 [nvars], page 230)

`opposite` opposite ring (see Section 7.3.20 [opposite], page 347)

`setring` set a new basering (see Section 5.1.137 [setring], page 255)

7.2.6 resolution (plural)

The type resolution is intended as an intermediate representation which internally retains additional information obtained during computation of resolutions. It furthermore enables the use of partial results to compute, for example, Betti numbers or minimal resolutions. Like ideals and modules, a resolution can only be defined w.r.t. a basering.

Note: to access the elements of a resolution, it has to be assigned to a list. This assignment also completes computations and may therefore take time, (resp. an access directly with the brackets `[ , ]` causes implicitly a cast to a list).

7.2.6.1 resolution declarations (plural)

Syntax: `resolution` name = resolution_expression ;

Purpose: defines a resolution.

Default: none

Example:

```
ring r=0,(x,y,z),dp;
matrix D[3][3];
D[1,2]=z;
def R=nc_algebra(1,D); // it is a Heisenberg algebra
setring R;
ideal i=z2+z,x+y;
resolution re=nres(i,0);
re;
↦  1      2      1
↦ R <--  R <--  R
↦
↦ 0      1      2
↦ resolution not minimized yet
↦
list l = re;
l;
↦ [1]:
↦    _[1]=z2+z
↦    _[2]=x+y
↦ [2]:
↦    _[1]=z2*gen(2)-x*gen(1)-y*gen(1)+z*gen(2)
↦ [3]:
↦    _[1]=0
print(matrix(l[2]));
↦ -x-y,
↦ z2+z
print(module(transpose(matrix(l[2]))*transpose(matrix(l[1]))));  // check that product is zero
↦ 0
```

7.2.6.2 resolution expressions (plural)

A resolution expression is:

1. an identifier of type resolution
2. a function returning a resolution
3. a type cast to resolution from a list of ideals, resp. modules.

7.2.6.3 resolution related functions (plural)

`betti` Betti numbers of a resolution (see Section 7.3.1 [betti (plural)], page 329)

`minres` minimizes a free resolution (see Section 7.3.13 [minres (plural)], page 339)

`mres` computes a minimal free resolution of an ideal resp. module and a minimal set of generators of the given ideal resp. module (see Section 7.3.15 [mres (plural)], page 341)

`nres` computes a free resolution of an ideal resp. module M which is minimized from the second module on (see Section 7.3.18 [nres (plural)], page 345)

7.2.7 ring (plural)

Rings are used to describe properties of polynomials, ideals etc. Almost all computations in Plural require a basering. For a detailed description of the concept of rings see Section 3.3 [Rings and orderings], page 30.

Note: Plural usually works with global orderings (see Section 7.1 [PLURAL], page 312) but one can use certain local once when graded commutative rings are being used.

7.2.7.1 ring declarations (plural)

Syntax: `ring` name `=` `(` coefficient_field `)`, `(` names_of_ring_variables `)`, `(` ordering `);`

Default: `32003,(x,y,z),(dp,C);`

Purpose: declares a ring and sets it as the actual basering.

The coefficient_field is given by one of the following:

1. a non-negative int_expression less or equal 2147483647.
2. an expression_list of an int_expression and one or more names.
3. the name `real`.
4. an expression_list of the name `real` and an int_expression.
5. an expression_list of the name `complex`, an optional int_expression and a name.

'names_of_ring_variables' must be a list of names or indexed names.

'ordering' is a list of block orderings where each block ordering is either

1. `lp`, `dp`, `Dp`, optionally followed by a size parameter in parentheses.
2. `wp`, `Wp`, or `a` followed by a weight vector given as an intvec_expression in parentheses.
3. `M` followed by an intmat_expression in parentheses.
4. `c` or `C`.

As long as all non-commuting variables are global, any ordering may be used. In graded commutative algebras, one may also use `ls, ds, Ds, ws`, and `Ws`.

If one of coefficient_field, names_of_ring_variables, and ordering consists of only one entry, the parentheses around this entry may be omitted.

In order to create a non-commutative structure over a commutative ring, use Section 7.3.16 [nc_algebra], page 342.

7.2.7.2 ring operations (plural)

`+` construct a tensor product $C = A \otimes_{\mathbf{K}} B$ of two G-algebras A and B over the ground field. Let, e.g.,

$A = k_1\langle x_1, \ldots, x_n \mid \{x_j x_i = c_{ij} \cdot x_i x_j + d_{ij}\}, 1 \leq i < j \leq n\rangle$, and $B = k_2\langle y_1, \ldots, y_m \mid \{y_j y_i = q_{ij} \cdot y_i y_j + r_{ij}\}, 1 \leq i < j \leq m\rangle$

be two G -algebras, then C is defined to be the algebra

$C = K\langle x_1, \ldots, x_n, y_1, \ldots, y_m \mid \{x_j x_i = c_{ij} \cdot x_i x_j + d_{ij}, 1 \leq i < j \leq n\}, \{y_j y_i = q_{ij} \cdot y_i y_j + r_{ij}, 1 \leq i < j \leq m\}, \{y_j x_i = x_i y_j, 1 \leq j \leq m, 1 \leq i \leq n\}\rangle$.

Concerning the ground fields k_1 resp. k_2 of A resp. B , take the following guidelines for $A \otimes_{\mathbf{K}} B$ into consideration:

- Neither k_1 nor k_2 may be R or C .
- If the characteristic of k_1 and k_2 differs, then one of them must be Q .
- At most one of k_1 and k_2 may have parameters.
- If one of k_1 and k_2 is an algebraic extension of Z/p it may not be defined by a `charstr` of type `(p^n,a)`.

One can create a ring using `ring(list)`, see also `ringlist`.

Example:

```
LIB "ncalg.lib";
def a = makeUsl2();       // U(sl_2) in e,f,h presentation
ring W0 = 0,(x,d),dp;
def W = Weyl();              // 1st Weyl algebra in x,d
def S = a+W;
setring S;
S;
↦ // coefficients: QQ
↦ // number of vars : 5
↦ //        block   1 : ordering dp
↦ //                  : names    e f h
↦ //        block   2 : ordering dp
↦ //                  : names    x d
↦ //        block   3 : ordering C
↦ // noncommutative relations:
↦ //    fe=ef-h
↦ //    he=eh+2e
↦ //    hf=fh-2f
↦ //    dx=xd+1
```

7.2.7.3 ring related functions (plural)

`charstr` description of the coefficient field of a ring (see Section 5.1.7 [charstr], page 160) `envelope` enveloping ring (see Section 7.3.6 [envelope], page 334)

`npars` number of ring parameters (see Section 5.1.102 [npars], page 228)

`nvars` number of ring variables (see Section 5.1.106 [nvars], page 230)

`opposite` opposite ring (see Section 7.3.20 [opposite], page 347)

`ordstr` monomial ordering of a ring (see Section 5.1.110 [ordstr], page 235)

`parstr` names of all ring parameters or the name of the n-th ring parameter (see Section 5.1.113 [parstr], page 236)

`qring` quotient ring (see Section 7.2.5 [qring (plural)], page 324)

`ringlist` decomposes a ring into a list of its components (see Section 7.3.24 [ringlist (plural)], page 352)

`setring` set a new basering (see Section 5.1.137 [setring], page 255)

`varstr` names of all ring variables or the name of the n-th ring variable (see Section 5.1.162 [varstr], page 280)

7.3 Functions (plural)

This chapter gives a complete reference of all functions and commands of the Plural kernel, i.e. all built-in commands (for the Plural libraries see Section 7.7 [Non-commutative libraries], page 370).

The general syntax of a function is

[target =] function_name (<arguments>);

Note, that both **Control structures** and **System variables** of Plural are the same as of Singular (see Section 5.2 [Control structures], page 284, Section 5.3 [System variables], page 297).

7.3.1 betti (plural)

Syntax: `betti (` list_expression `)`
`betti (` resolution_expression `)`
`betti (` list_expression `,` int_expression `)`
`betti (` resolution_expression `,` int_expression `)`

Type: intmat

Note: in the non-commutative case, computing Betti numbers makes sense only if the basering R has homogeneous relations. The output of the command can be pretty-printed using `print( , ''betti'')`, i.e., with "betti" as second argument; see below example.

Purpose: with 1 argument: computes the graded Betti numbers of a minimal resolution of R^n/M, if R denotes the basering and M a homogeneous submodule of R^n and the argument represents a resolution of R^n/M.
The entry d of the intmat at place (i, j) is the minimal number of generators in degree i+j of the j-th syzygy module (= module of relations) of R^n/M (the 0th (resp. 1st) syzygy module of R^n/M is R^n (resp. M)). The argument is considered to be the result of a `mres` or `nres` command. This implies that a zero is only allowed (and counted) as a generator in the first module.
For the computation `betti` uses only the initial monomials. This could lead to confusing results for a non-homogeneous input.

If the optional second argument is non-zero, the Betti numbers will be minimized.

Example:

```
int i;int N=2;
ring r=0,(x(1..N),d(1..N),q(1..N)),Dp;
matrix D[3*N][3*N];
for (i=1;i<=N;i++)
{ D[i,N+i]=q(i)^2; }
def W=nc_algebra(1,D); setring W;
// this algebra is a kind of homogenized Weyl algebra
W;
↦ // coefficients: QQ
↦ // number of vars : 6
↦ //        block   1 : ordering Dp
↦ //                  : names    x(1) x(2) d(1) d(2) q(1) q(2)
↦ //        block   2 : ordering C
↦ // noncommutative relations:
↦ //    d(1)x(1)=x(1)*d(1)+q(1)^2
↦ //    d(2)x(2)=x(2)*d(2)+q(2)^2
ideal I = x(1),x(2),d(1),d(2),q(1),q(2);
option(redSB);
option(redTail);
resolution R = mres(I,0);
// thus R will be the full length minimal resolution
print(betti(R),"betti");
↦            0     1     2     3     4     5     6
↦ ------------------------------------------------
↦     0:     1     6    15    20    15     6     1
↦ ------------------------------------------------
↦ total:     1     6    15    20    15     6     1
↦
```

7.3.2 bracket

Syntax: `bracket` (poly_expression, poly_expression)

Type: poly

Purpose: Computes the Lie bracket `[p,q]=pq-qp` of the first polynomial with the second. Uses special routines, based on the Leibniz rule.

Example:

```
ring r=(0,Q),(x,y,z),Dp;
minpoly=Q^2-Q+1;
matrix C[3][3];  matrix D[3][3];
C[1,2]=Q2;    C[1,3]=1/Q2;  C[2,3]=Q2;
D[1,2]=-Q*z;  D[1,3]=1/Q*y; D[2,3]=-Q*x;
def R=nc_algebra(C,D); setring R; R;
↦ // coefficients: QQ[Q]/(Q2-Q+1)
↦ // number of vars : 3
↦ //        block   1 : ordering Dp
↦ //                  : names    x y z
↦ //        block   2 : ordering C
↦ // noncommutative relations:
↦ //    yx=(Q-1)*xy+(-Q)*z
↦ //    zx=(-Q)*xz+(-Q+1)*y
```

```
↦ //    zy=(Q-1)*yz+(-Q)*x
// this is a quantum deformation of U(so_3),
// where Q is a 6th root of unity
poly p=Q^4*x2+y2+Q^4*z2+Q*(1-Q^4)*x*y*z;
// p is the central element of the algebra
p=p^3; // any power of a central element is central
poly q=(x+Q*y+Q^2*z)^4;
// take q to be some big noncentral element
size(q); // check how many monomials are in big polynomial q
↦ 28
bracket(p,q); // check p*q=q*p
↦ 0
// a more common behaviour of the bracket follows:
bracket(x+Q*y+Q^2*z,z);
↦ (Q+1)*xz+(Q+1)*yz+(Q-1)*x+(Q-1)*y
```

7.3.3 dim (plural)

Syntax: `dim` (ideal_expression)
`dim` (module_expression)

Type: int

Purpose: computes the Gelfand-Kirillov dimension of the ideal, resp. module, generated by the leading monomials of the given generators of the ideal, resp. module. This is also the dimension of the ideal resp. submodule, if it is represented by a left Groebner basis.

Note: The dimension of a submodule of a free module is defined to be the Gelfand-Kirillov dimension of the left module with the presentation via given submodule.
The computed Gelfand-Kirillov dimension is taken relative to the ground field. In order to compute the complete Gelfand-Kirillov dimension, one has to add the transcendence degree of the ground field.

Example:

```
  ring r=0,(x,y,Dx,Dy),dp;
  matrix M[4][4]; M[1,3]=1;M[2,4]=1;
  def R = nc_algebra(1,M); // 2nd Weyl algebra
  setring R;
  dim(std(0)); // the GK dimension of the ring itself
↦ 4
  ideal I=x*Dy^2-2*y*Dy^2+2*Dy, Dx^3+3*Dy^2;
  dim(std(I)); // the GK dimension of the module R/I
↦ 2
  module T = (x*Dx -2)*gen(1), Dx^3*gen(1), (y*Dy +3)*gen(2);
  dim(std(T)); // the GK dimension of the module  R^2/T
↦ 3
```

See Section 4.5 [ideal], page 79; Section 4.13 [module], page 111; Section 5.1.147 [std], page 266; Section 5.1.163 [vdim], page 280.

7.3.4 division (plural)

Syntax: `division` (ideal_expression, ideal_expression)
`division` (module_expression, module_expression)

`division` (ideal_expression, ideal_expression, int_expression)
`division` (module_expression, module_expression, int_expression)
`division` (ideal_expression, ideal_expression, int_expression, intvec_expression)
`division` (module_expression, module_expression, int_expression, intvec_expression)

Type: list

Purpose: `division` computes a left division with remainder. For two left ideals resp. modules `M` (first argument) and `N` (second argument), it returns a list `T,R,U` where `T` is a matrix, `R` is a left ideal resp. a module, and `U` is a diagonal matrix of units such that `transpose(U)*transpose(matrix(M))=transpose(T)*transpose(matrix(N)) + transpose(matrix(R))`. From this data one gets a left standard representation for the left normal form `R` of `M` with respect to a left Groebner basis of `N`. `division` uses different algorithms depending on whether `N` is represented by a Groebner basis. For a GR-algebra, the matrix `U` is the identity matrix. A matrix `T` as above is also computed by `lift`.
For additional arguments `n` (third argument) and `w` (fourth argument), `division` returns a list `T,R` as above such that `transpose(matrix(M))=transpose(T)*transpose(matrix(N)) + transpose(matrix(R))` is a left standard representation for the left normal form `R` of `M` with respect to `N` up to weighted degree `n` with respect to the weight vector `w`. The weighted degree of `T` and `R` respect to `w` is at most `n`. If the weight vector `w` is not given, `division` uses the standard weight vector `w=1,...,1`.

Example:

```
LIB "dmod.lib";
ring r = 0,(x,y),dp;
poly f = x^3+xy;
def S = Sannfs(f); setring S; // compute the annihilator of f^s
LD; // is not a Groebner basis yet!
↦ LD[1]=3*x^2*Dy-x*Dx+y*Dy
↦ LD[2]=x*Dx+2*y*Dy-3*s
poly f = imap(r,f);
poly P = f*Dx-s*diff(f,x);
division(P,LD); // so P is in the ideal via the cofactors in _[1]
↦ [1]:
↦    _[1,1]=-2/3*y
↦    _[2,1]=x^2+1/3*y
↦ [2]:
↦    _[1]=0
↦ [3]:
↦    _[1,1]=1
ideal I = LD, f; // consider a bigger ideal
list L = division(s^2, I); // the normal form is -2s-1
L;
↦ [1]:
↦    _[1,1]=2/3*x^2*Dy-1/3*x*Dx+2/3*s+1/3
↦    _[2,1]=2/3*x^2*Dy-1/3*x*Dx-1/3*s-2/3
↦    _[3,1]=-2*x*Dy^2+Dx*Dy
↦ [2]:
↦    _[1]=-2*s-1
↦ [3]:
```

```
↦    _[1,1]=1
// now we show that the formula above holds
matrix M[1][1] = s^2; matrix N = matrix(I);
matrix T = matrix(L[1]); matrix R = matrix(L[2]); matrix U  = matrix(L[3]);
// the formula must return zero:
transpose(U)*transpose(M) - transpose(T)*transpose(N) - transpose(R);
↦ _[1,1]=0
```

See Section 4.5 [ideal], page 79; Section 5.1.78 [lift], page 209; Section 4.13 [module], page 111; Section 4.16 [poly], page 118; Section 4.21 [vector], page 131.

7.3.5 eliminate (plural)

Syntax: `eliminate` (ideal_expression`,` product_of_ring_variables)
`eliminate` (module_expression`,` product_of_ring_variables)

Type: the same as the type of the first argument

Purpose: eliminates variables occurring as factors of the second argument from an ideal (resp. a submodule of a free module), by intersecting it (resp. each component of the submodule) with the subring not containing these variables.

Note: `eliminate` neither needs a special ordering on the basering nor a Groebner basis as input. Moreover, `eliminate` does not work in non-commutative quotients.

Remark: in a non-commutative algebra, not every subset of a set of variables generates a proper subalgebra. But if it is so, there may be cases, when no elimination is possible. In these situations error messages will be reported.

Example:

```
ring r=0,(e,f,h,a),Dp;
matrix d[4][4];
d[1,2]=-h; d[1,3]=2*e; d[2,3]=-2*f;
def R=nc_algebra(1,d); setring R;
// this algebra is U(sl_2), tensored with K[a] over K
option(redSB);
option(redTail);
poly  p = 4*e*f+h^2-2*h - a;
// p is a central element with parameter
ideal I = e^3, f^3, h^3-4*h, p; // take this ideal
// and intersect I with the ring K[a]
ideal J = eliminate(I,e*f*h);
// if we want substitute 'a' with a value,
// it has to be a root of this polynomial
J;
↦ J[1]=a3-32a2+192a
// now we try to eliminate h,
// that is we intersect I with the subalgebra S,
// generated by e and f.
// But S is not closed in itself, since f*e-e*f=-h !
// the next command will definitely produce an error
eliminate(I,h);
↦    ? no elimination is possible: subalgebra is not admissible
↦    ? error occurred in or before ./examples/eliminate_(plural).sing l\
   ine 21: `eliminate(I,h);`
```

```
// since a commutes with e,f,h, we can eliminate it:
eliminate(I,a);
↦ _[1]=h3-4h
↦ _[2]=fh2-2fh
↦ _[3]=f3
↦ _[4]=eh2+2eh
↦ _[5]=2efh-h2-2h
↦ _[6]=e3
```

See Section 7.2.1 [ideal (plural)], page 313; Section 7.2.3 [module (plural)], page 319; Section 7.3.26 [std (plural)], page 355.

7.3.6 envelope

Syntax: `envelope` (ring_name)

Type: ring

Purpose: creates an enveloping algebra of a given algebra, that is $A^{env} = A \otimes_K A^{opp}$, where A^{opp} is the opposite algebra of A.

Remark: You have to activate the ring with the `setring` command. For the presentation, see explanation of `opposite` in Section 7.3.20 [opposite], page 347.

```
LIB "ncalg.lib";
def A = makeUsl2();
setring A; A;
↦ // coefficients: QQ
↦ // number of vars : 3
↦ //        block   1 : ordering dp
↦ //                  : names    e f h
↦ //        block   2 : ordering C
↦ // noncommutative relations:
↦ //    fe=ef-h
↦ //    he=eh+2e
↦ //    hf=fh-2f
def Aenv = envelope(A);
setring Aenv;
Aenv;
↦ // coefficients: QQ
↦ // number of vars : 6
↦ //        block   1 : ordering dp
↦ //                  : names    e f h
↦ //        block   2 : ordering a
↦ //                  : names    H F E
↦ //                  : weights  1 1 1
↦ //        block   3 : ordering ls
↦ //                  : names    H F E
↦ //        block   4 : ordering C
↦ // noncommutative relations:
↦ //    fe=ef-h
↦ //    he=eh+2e
↦ //    hf=fh-2f
↦ //    FH=HF-2F
↦ //    EH=HE+2E
↦ //    EF=FE-H
```

See Section 7.3.19 [oppose], page 346; Section 7.3.20 [opposite], page 347.

7.3.7 fetch (plural)

Syntax: `fetch` (ring_name, name)

Type: number, poly, vector, ideal, module, matrix or list (the same type as the second argument)

Purpose: maps objects between rings. `fetch` is the identity map between rings and qrings, the i-th variable of the source ring is mapped to the i-th variable of the basering. The coefficient fields must be compatible. (See Section 7.2.2 [map (plural)], page 317 for a description of possible mappings between different ground fields).
`fetch` offers a convenient way to change variable names or orderings, or to map objects from a ring to a quotient ring of that ring or vice versa.

Note: Compared with `imap`, `fetch` uses the position of the ring variables, not their names.

Example:

```
LIB "ncalg.lib";
def Usl2 = makeUsl2(); // this algebra is U(sl_2)
setring Usl2;
option(redSB);
option(redTail);
poly  C  = 4*e*f+h^2-2*h; // the central element of Usl2
ideal I  = e^3,f^3,h^3-4*h;
ideal J  = twostd(I);
 // print a compact presentation of J:
print(matrix(ideal(J[1..5]))); // first 5 generators
↦ h3-4h,fh2-2fh,eh2+2eh,f2h-2f2,2efh-h2-2h
print(matrix(ideal(J[6..size(J)]))); // last generators
↦ e2h+2e2,f3,ef2-fh,e2f-eh-2e,e3
ideal QC = twostd(C-8);
qring Q  = QC;
ideal QJ = fetch(Usl2,J);
QJ = std(QJ);
// thus QJ is the image of I in the factor-algebra QC
print(matrix(QJ)); // print QJ compactly
↦ h3-4h,fh2-2fh,eh2+2eh,f2h-2f2,e2h+2e2,f3,e3
```

See Section 7.3.8 [imap (plural)], page 335; Section 7.2.2 [map (plural)], page 317; Section 7.2.5 [qring (plural)], page 324; Section 7.2.7 [ring (plural)], page 327.

7.3.8 imap (plural)

Syntax: `imap` (ring_name, name)

Type: number, poly, vector, ideal, module, matrix or list (the same type as the second argument)

Purpose: identity map on common subrings. `imap` is the map between rings and qrings with compatible ground fields which is the identity on variables and parameters of the same name and 0 otherwise. (See Section 7.2.2 [map (plural)], page 317 for a description of possible mappings between different ground fields). Useful for mappings from a homogenized ring to the original ring or for mappings from/to rings with/without parameters. Compared with `fetch`, `imap` uses the names of variables and parameters. **Unlike `map` and `fetch`, `imap` can map parameters to variables.**

Example:

```
LIB "ncalg.lib";
ring ABP=0,(p4,p5,a,b),dp; //  a commutative ring
def Usl3 = makeUsl(3);
def BIG  = Usl3+ABP;
setring BIG;
poly P4 = 3*x(1)*y(1)+3*x(2)*y(2)+3*x(3)*y(3);
P4 = P4 +h(1)^2+h(1)*h(2)+h(2)^2-3*h(1)-3*h(2);
// P4 is a central element of Usl3 of degree 2
poly P5 = 4*x(1)*y(1) + h(1)^2 - 2*h(1);
// P5 is a central element of the subalgebra of U(sl_3),
// generated by x(1),y(1),h(1)
ideal J = x(1),x(2),h(1)-a,h(2)-b;
// we are interested in the module U(sl_3)/J,
// which depends on parameters a,b
ideal I = p4-P4, p5-P5;
ideal K = I, J;
ideal E = eliminate(K,x(1)*x(2)*x(3)*y(1)*y(2)*y(3)*h(1)*h(2));
E; // this is the ideal of central characters in ABP
↦ E[1]=a*b+b^2-p4+p5+a+3*b
↦ E[2]=a^2-p5+2*a
↦ E[3]=b^3+p4*a-p5*a-a^2-p4*b+3*b^2
// what are the characters on nonzero a,b?
ring abP = (0,a,b),(p4,p5),dp;
ideal abE = imap(BIG, E);
option(redSB);
option(redTail);
abE = std(abE);
// here come characters (indeed, we have only one)
// that is a maximal ideal in K[p4,p5]
abE;
↦ abE[1]=p5+(-a^2-2*a)
↦ abE[2]=p4+(-a^2-a*b-3*a-b^2-3*b)
```

See Section 7.3.7 [fetch (plural)], page 335; Section 7.2.2 [map (plural)], page 317; Section 7.2.5 [qring (plural)], page 324; Section 7.2.7 [ring (plural)], page 327.

7.3.9 intersect (plural)

Syntax: `intersect` (expression_list of ideal_expression)
`intersect` (expression_list of module_expression)

Type: ideal, resp. module

Purpose: computes the intersection of ideals, resp. modules.

Example:

```
ring r=0,(x,y),dp;
def R=nc_algebra(-1,0);  //anti-commutative algebra
setring R;
module M=[x,x],[y,0];
module N=[0,y^2],[y,x];
```

```
option(redSB);
module Res;
Res=intersect(M,N);
print(Res);
↦ y2, 0,
↦ -xy,xy2
kill r,R;
//--------------------------------
LIB "ncalg.lib";
ring r=0,(x,d),dp;
def RR=Weyl(); // make r into Weyl algebra
setring RR;
ideal I = x+d^2;
ideal J = d-1;
ideal H = intersect(I,J);
H;
↦ H[1]=d4+xd2-2d3-2xd+d2+x+2d-2
↦ H[2]=xd3+x2d-xd2+d3-x2+xd-2d2-x+1
```

7.3.10 kbase (plural)

Syntax: `kbase` (ideal_expression)
`kbase` (module_expression)
`kbase` (ideal_expression, int_expression)
`kbase` (module_expression, int_expression)

Type: the same as the input type of the first argument

Purpose: with one argument: computes the vector space basis of the factor-module that equals ring (resp. free module) modulo the ideal (resp. submodule), generated by the initial terms of the given generators.
If the factor-module is not of finite dimension, -1 is returned.

If the generators form a Groebner basis, this is the same as the vector space basis of the factor-module.

when called with two arguments: computes the part of a vector space basis of the respective quotient with degree (of monomials) equal to the second argument. Here, the quotient does not need to be finite dimensional.

Note: in the non-commutative case, a ring modulo an ideal has a ring stucture if and only if the ideal is two-sided. `kbase` respects module-grading given by the `isHomog` attribute of input modules.

Example:

```
ring r=0,(x,y,z),dp;
matrix d[3][3];
d[1,2]=-z;  d[1,3]=2x;  d[2,3]=-2y;
def R=nc_algebra(1,d); // this algebra is U(sl_2)
setring R;
ideal i=x2,y2,z2-1;
i=std(i);
print(matrix(i));  // print a compact presentation of i
↦ z2-1,yz-y,xz+x,y2,2xy-z-1,x2
kbase(i);
```

```
↦ _[1]=z
↦ _[2]=y
↦ _[3]=x
↦ _[4]=1
vdim(i);
↦ 4
ideal j=x,z-1;
j=std(j);
kbase(j,3);
↦ _[1]=y3
```

See Section 7.2.1 [ideal (plural)], page 313; Section 7.2.3 [module (plural)], page 319; Section 7.3.30 [vdim (plural)], page 358.

7.3.11 lift (plural)

Syntax: `lift` (ideal_expression, subideal_expression)
`lift` (module_expression, submodule_expression)

Type: matrix

Purpose: computes the (left) transformation matrix which expresses the (left) generators of a submodule in terms of the (left) generators of a module. Uses different algorithms for modules which are (resp. are not) represented by a Groebner basis.
More precisely, if `m` is the module, `sm` the submodule, and `T` the transformation matrix returned by lift, then `transpose(matrix(sm)) = transpose(T)*transpose(matrix(m))`. If `m` and `sm` are ideals, `ideal(sm) = ideal(transpose(T)*transpose(matrix(m)))`.

Note: Gives a warning if `sm` is not a submodule.

Example:

```
ring r = (0,a),(e,f,h),(c,dp);
matrix D[3][3];
D[1,2]=-h;  D[1,3]=2*e;  D[2,3]=-2*f;
def R=nc_algebra(1,D); // this algebra is a parametric U(sl_2)
setring R;
ideal i = e,h-a; // consider this parametric ideal
i = std(i);
print(matrix(i)); // print a compact presentation of i
↦ h+(-a),e
poly Z = 4*e*f+h^2-2*h; // a central element
Z = Z - NF(Z,i); // a central character
ideal j = std(Z);
j;
↦ j[1]=4*ef+h2-2*h+(-a2-2a)
matrix T = lift(i,j);
print(T);
↦ h+(a+2),
↦ 4*f
ideal tj = ideal(transpose(T)*transpose(matrix(i)));
size(ideal(matrix(j)-matrix(tj))); // test for 0
↦ 0
```

See Section 7.2.1 [ideal (plural)], page 313; Section 7.3.12 [liftstd (plural)], page 339; Section 7.2.3 [module (plural)], page 319.

7.3.12 liftstd (plural)

Syntax: `liftstd` (ideal_expression, matrix_name)
`liftstd` (module_expression, matrix_name)
`liftstd` (ideal_expression, matrix_name, module_name)
`liftstd` (module_expression, matrix_name, module_name)

Type: ideal or module

Purpose: returns a left Groebner basis of an ideal or module and a left transformation matrix from the given ideal, resp. module, to the Groebner basis.
That is, if `m` is the ideal or module, `sm` is the left Groebner basis of `m`, returned by `liftstd`, and `T` is a left transformation matrix, then `sm=module(transpose(transpose(T)*transpose(matrix(m))))`.
If `m` is an ideal, `sm=ideal(transpose(T)*transpose(matrix(m)))`.
In an optional third argument the left syzygy module will be returned.

Example:

```
LIB "ncalg.lib";
def A = makeUsl2();
setring A;  // this algebra is U(sl_2)
ideal i = e2,f;
option(redSB);
option(redTail);
matrix T;
ideal j = liftstd(i,T);
// the Groebner basis in a compact form:
print(matrix(j));
↦ f,2h2+2h,2eh+2e,e2
print(T);  // the transformation matrix
↦ 0,f2,        -f,1,
↦ 1,-e2f+4eh+8e,e2,0
ideal tj = ideal(transpose(T)*transpose(matrix(i)));
size(ideal(matrix(j)-matrix(tj))); // test for 0
↦ 0
module S; ideal k = liftstd(i,T,S); // the third argument
S = std(S); print(S); // the syzygy module
↦ -ef-2h+6,-f3,
↦ e3,      e2f2-6efh-6ef+6h2+18h+12
```

See Section 7.2.1 [ideal (plural)], page 313; Section 7.2.7 [ring (plural)], page 327; Section 7.3.26 [std (plural)], page 355.

7.3.13 minres (plural)

Syntax: `minres` (list_expression)

Type: list

Syntax: `minres` (resolution_expression)

Type: resolution

Purpose: minimizes a free resolution of an ideal or module given by the list_expression, resp. resolution_expression.

Example:

```
LIB "ncalg.lib";
def A = makeUsl2();
setring A; // this algebra is U(sl_2)
ideal i=e,f,h;
i=std(i);
resolution F=nres(i,0); F;
↦  1      3      3      1
↦ A <--  A <--  A <--  A
↦
↦ 0      1      2      3
↦ resolution not minimized yet
↦
list lF = F; lF;
↦ [1]:
↦    _[1]=h
↦    _[2]=f
↦    _[3]=e
↦ [2]:
↦    _[1]=f*gen(1)-h*gen(2)-2*gen(2)
↦    _[2]=e*gen(1)-h*gen(3)+2*gen(3)
↦    _[3]=e*gen(2)-f*gen(3)-gen(1)
↦ [3]:
↦    _[1]=e*gen(1)-f*gen(2)+h*gen(3)
print(betti(lF), "betti");
↦            0     1     2     3
↦ ------------------------------
↦     0:     1     -     3     1
↦ ------------------------------
↦ total:     1     0     3     1
↦
resolution MF=minres(F);  MF;
↦  1      2      2      1
↦ A <--  A <--  A <--  A
↦
↦ 0      1      2      3
↦
list lMF = F; lMF;
↦ [1]:
↦    _[1]=f
↦    _[2]=e
↦ [2]:
↦    _[1]=-ef*gen(1)+f2*gen(2)+2h*gen(1)+2*gen(1)
↦    _[2]=-e2*gen(1)+ef*gen(2)+h*gen(2)-2*gen(2)
↦ [3]:
↦    _[1]=e*gen(1)-f*gen(2)
print(betti(lMF), "betti");
↦            0     1     2     3
↦ ------------------------------
↦     0:     1     -     -     -
↦     1:     -     -     2     1
↦ ------------------------------
↦ total:     1     0     2     1
↦
```

See Section 7.3.15 [mres (plural)], page 341; Section 7.3.18 [nres (plural)], page 345.

7.3.14 modulo (plural)

Syntax: `modulo` (ideal_expression, ideal_expression)
`modulo` (module_expression, module_expression)

Type: module

Purpose: `modulo(h1,h2)` represents $h_1/(h_1 \cap h_2) \cong (h_1+h_2)/h_2$, where h_1 and h_2 are considered as submodules of the same free module R^l (l=1 for ideals).
Let H_1 (resp. H_2) be the matrix of size $l \times k$ (resp. $l \times m$), having the generators of h_1 (resp. h_2) as columns.
Then $h_1/(h_1 \cap h_2) \cong R^k/ker(\overline{H_1})$, where $\overline{H_1} : R^k \to R^l/Im(H_2) = R^l/h_2$ is the induced map given by H_1.
`modulo(h1,h2)` returns generators of the kernel of this induced map.

Note: `If, for at least one of`
h_1 or h_2, the attribute `isHomog` is st, then `modulo(h1,h2)` also sets this attribute (if the weights are compatible).

Example:

```
LIB "ncalg.lib";
def A = makeUsl2();
setring A; // this algebra is U(sl_2)
option(redSB);
option(redTail);
ideal H2 = e2,f2,h2-1;
H2 = twostd(H2);
print(matrix(H2)); // print H2 in a compact form
↦ h2-1,fh-f,eh+e,f2,2ef-h-1,e2
ideal H1 = std(e);
ideal T = modulo(H1,H2);
T = NF(std(H2+T),H2);
T = std(T);
T;
↦ T[1]=h-1
↦ T[2]=e
```

See also Section 7.3.28 [syz (plural)], page 357.

7.3.15 mres (plural)

Syntax: `mres` (ideal_expression, int_expression)
`mres` (module_expression, int_expression)

Type: resolution

Purpose: computes a minimal free resolution of an ideal or module M with the Groebner basis method. More precisely, let A=`matrix`(M), then `mres` computes a free resolution of $coker(A) = F_0/M$

$$... \longrightarrow F_2 \xrightarrow{A_2} F_1 \xrightarrow{A_1} F_0 \longrightarrow F_0/M \longrightarrow 0,$$

where the columns of the matrix A_1 are a (possibly) minimal set of generators of M .
If the int expression k is not zero, then the computation stops after k steps and returns

a resolution consisting of modules $M_i =$ `module`(A_i), $i = 1 \ldots k$.
`mres(M,0)` returns a resolution consisting of at most n+2 modules, where n is the number of variables of the basering. Let `list L=mres(M,0);` then `L[1]` consists of a minimal set of generators `M`, `L[2]` consists of a minimal set of generators for the first syzygy module of `L[1]`, etc., until `L[p+1]`, such that `L[i]` $\neq 0$ for $i \leq p$, but `L[p+1]` (the first syzygy module of `L[p]`) is 0 (if the basering is not a qring).

Note: Accessing single elements of a resolution may require that some partial computations have to be finished and may therefore take some time. Hence, assigning right away to a list is the recommended way to do it.

Example:

```
LIB "ncalg.lib";
def A = makeUsl2();
setring A; // this algebra is U(sl_2)
option(redSB);
option(redTail);
ideal i = e,f,h;
i = std(i);
resolution M=mres(i,0);
M;
↦  1      2      2      1
↦ A <--  A <--  A <--  A
↦
↦ 0      1      2      3
↦
list l = M; l;
↦ [1]:
↦    _[1]=f
↦    _[2]=e
↦ [2]:
↦    _[1]=ef*gen(1)-f2*gen(2)-2h*gen(1)-2*gen(1)
↦    _[2]=e2*gen(1)-ef*gen(2)-h*gen(2)+2*gen(2)
↦ [3]:
↦    _[1]=e*gen(1)-f*gen(2)
// see the exactness at this point
size(ideal(transpose(l[2])*transpose(l[1])));
↦ 0
print(matrix(M[3]));
↦ e,
↦ -f
// see the exactness at this point
size(ideal(transpose(l[3])*transpose(l[2])));
↦ 0
```

See Section 7.2.1 [ideal (plural)], page 313; Section 7.3.13 [minres (plural)], page 339; Section 7.2.3 [module (plural)], page 319; Section 7.3.18 [nres (plural)], page 345.

7.3.16 nc_algebra

Syntax:

`nc_algebra(` matrix_expression C, matrix_expression D `)`
`nc_algebra(` number_expression n, matrix_expression D `)`

`nc_algebra(` matrix_expression C, poly_expression p `)`
`nc_algebra(` number_expression n, poly_expression p `)`

Type: ring

Purpose: Executed in the basering `r`, say, in k variables $x_1, \ldots, x_k$, `nc_algebra` creates and returns the non-commutative extension of `r` subject to relations $\{x_j x_i = c_{ij} \cdot x_i x_j + d_{ij}, 1 \leq i < j \leq k\}$, where c_{ij} and d_{ij} must be put into two strictly upper triangular matrices `C` with entries c_{ij} from the ground field of `r` and `D` with (commutative) polynomial entries d_{ij} from `r`. See all the details in Section 7.4.1 [G-algebras], page 359.
If $\forall i < j$, $c_{ij} = n$, one can input the number `n` instead of matrix `C`.
If $\forall i < j$, $d_{ij} = p$, one can input the polynomial `p` instead of matrix `D`.

Note: `The returned ring should be activated afterwards, using the command setring.`

Remark: At present, PLURAL does not check the non-degeneracy conditions (see Section 7.4.1 [G-algebras], page 359) while setting an algebra.

Example:

```
LIB "nctools.lib";
// ------- first example: C, D are matrices --------
ring r1 = (0,Q),(x,y,z),Dp;
minpoly = rootofUnity(6);
matrix C[3][3];
matrix D[3][3];
C[1,2]=Q2;   C[1,3]=1/Q2;  C[2,3]=Q2;
D[1,2]=-Q*z; D[1,3]=1/Q*y; D[2,3]=-Q*x;
def S=nc_algebra(C,D);
// this algebra is a quantum deformation U'_q(so_3),
// where Q is a 6th root of unity
setring S;S;
↦ // coefficients: QQ[Q]/(Q2-Q+1)
↦ // number of vars : 3
↦ //        block   1 : ordering Dp
↦ //                  : names    x y z
↦ //        block   2 : ordering C
↦ // noncommutative relations:
↦ //    yx=(Q-1)*xy+(-Q)*z
↦ //    zx=(-Q)*xz+(-Q+1)*y
↦ //    zy=(Q-1)*yz+(-Q)*x
kill r1,S;
// ----- second example: number n=1, D is a matrix
ring r2=0,(Xa,Xb,Xc,Ya,Yb,Yc,Ha,Hb),dp;
matrix d[8][8];
d[1,2]=-Xc; d[1,4]=-Ha; d[1,6]=Yb;  d[1,7]=2*Xa;
d[1,8]=-Xa; d[2,5]=-Hb; d[2,6]=-Ya; d[2,7]=-Xb;
d[2,8]=2*Xb; d[3,4]=Xb; d[3,5]=-Xa; d[3,6]=-Ha-Hb;
d[3,7]=Xc;   d[3,8]=Xc; d[4,5]=Yc; d[4,7]=-2*Ya;
d[4,8]=Ya;  d[5,7]=Yb; d[5,8]=-2*Yb;
d[6,7]=-Yc; d[6,8]=-Yc;
def S=nc_algebra(1,d);  // this algebra is U(sl_3)
setring S;S;
↦ // coefficients: QQ
↦ // number of vars : 8
↦ //        block   1 : ordering dp
```

```
↦ //                  : names    Xa Xb Xc Ya Yb Yc Ha Hb
↦ //        block   2 : ordering C
↦ // noncommutative relations:
↦ //    XbXa=Xa*Xb-Xc
↦ //    YaXa=Xa*Ya-Ha
↦ //    YcXa=Xa*Yc+Yb
↦ //    HaXa=Xa*Ha+2*Xa
↦ //    HbXa=Xa*Hb-Xa
↦ //    YbXb=Xb*Yb-Hb
↦ //    YcXb=Xb*Yc-Ya
↦ //    HaXb=Xb*Ha-Xb
↦ //    HbXb=Xb*Hb+2*Xb
↦ //    YaXc=Xc*Ya+Xb
↦ //    YbXc=Xc*Yb-Xa
↦ //    YcXc=Xc*Yc-Ha-Hb
↦ //    HaXc=Xc*Ha+Xc
↦ //    HbXc=Xc*Hb+Xc
↦ //    YbYa=Ya*Yb+Yc
↦ //    HaYa=Ya*Ha-2*Ya
↦ //    HbYa=Ya*Hb+Ya
↦ //    HaYb=Yb*Ha+Yb
↦ //    HbYb=Yb*Hb-2*Yb
↦ //    HaYc=Yc*Ha-Yc
↦ //    HbYc=Yc*Hb-Yc
kill r2,S;
// ---- third example: C is a matrix, p=0 is a poly
ring r3=0,(a,b,c,d),lp;
matrix c[4][4];
c[1,2]=1; c[1,3]=3; c[1,4]=-2;
c[2,3]=-1; c[2,4]=-3; c[3,4]=1;
def S=nc_algebra(c,0); // it is a quasi--commutative algebra
setring S;S;
↦ // coefficients: QQ
↦ // number of vars : 4
↦ //        block   1 : ordering lp
↦ //                  : names    a b c d
↦ //        block   2 : ordering C
↦ // noncommutative relations:
↦ //    ca=3ac
↦ //    da=-2ad
↦ //    cb=-bc
↦ //    db=-3bd
kill r3,S;
// -- fourth example : number n = -1, poly p = 3w
ring r4=0,(u,v,w),dp;
def S=nc_algebra(-1,3w);
setring S;S;
↦ // coefficients: QQ
↦ // number of vars : 3
↦ //        block   1 : ordering dp
↦ //                  : names    u v w
↦ //        block   2 : ordering C
↦ // noncommutative relations:
```

```
↦ //    vu=-uv+3w
↦ //    wu=-uw+3w
↦ //    wv=-vw+3w
kill r4,S;
```

See also Section 7.7.12 [ncalg_lib], page 513; Section 7.7.16 [nctools_lib], page 571; Section 7.7.18 [qmatrix_lib], page 591.

7.3.17 ncalgebra

Syntax:
`ncalgebra(` matrix_expression C, matrix_expression D `)`
`ncalgebra(` number_expression n, matrix_expression D `)`
`ncalgebra(` matrix_expression C, poly_expression p `)`
`ncalgebra(` number_expression n, poly_expression p `)`

Type: none

Purpose: Works like Section 7.3.16 [nc_algebra], page 342 but changes the basering.

Remark: This function is **depreciated** and should be substituted by `nc_algebra`, since it violates the general Singular policy: only Section 4.19 [ring], page 125 and Section 5.1.137 [setring], page 255 can change the basering. More concretely, replace by `def A = nc_algebra(C, D); setring A;` which will additionally introduce a new ring. Afterwards, some objects may have to be mapped into the new ring.

See also Section 7.3.16 [nc_algebra], page 342; Section 7.7.16 [nctools_lib], page 571.

7.3.18 nres (plural)

Syntax: `nres (` ideal_expression`,` int_expression `)`
`nres (` module_expression`,` int_expression `)`

Type: resolution

Purpose: computes a free resolution of an ideal or module which is minimized from the second module on (by the Groebner basis method).

Note: `Assigning a resolution to a list is the best choice of usage. The resolution may be minimized by using the`
command `minres`. Use the command `betti` to compute Betti numbers.

Example:

```
LIB "ncalg.lib";
def A = makeUsl2();
setring A; // this algebra is U(sl_2)
option(redSB);
option(redTail);
ideal i = e,f,h;
i = std(i);
resolution F=nres(i,0); F;
↦  1      3      3      1
↦ A <--  A <--  A <--  A
↦
↦ 0      1      2      3
↦ resolution not minimized yet
```

```
↦
list l = F; l;
↦ [1]:
↦    _[1]=h
↦    _[2]=f
↦    _[3]=e
↦ [2]:
↦    _[1]=f*gen(1)-h*gen(2)-2*gen(2)
↦    _[2]=e*gen(1)-h*gen(3)+2*gen(3)
↦    _[3]=e*gen(2)-f*gen(3)-gen(1)
↦ [3]:
↦    _[1]=e*gen(1)-f*gen(2)+h*gen(3)
// see the exactness at this point:
size(ideal(transpose(l[2])*transpose(l[1])));
↦ 0
// see the exactness at this point:
size(ideal(transpose(l[3])*transpose(l[2])));
↦ 0
print(betti(l), "betti");
↦            0     1     2     3
↦ ------------------------------
↦     0:     1     -     3     1
↦ ------------------------------
↦ total:     1     0     3     1
↦
print(betti(minres(l)), "betti");
↦            0     1     2     3
↦ ------------------------------
↦     0:     1     -     -     -
↦     1:     -     -     2     1
↦ ------------------------------
↦ total:     1     0     2     1
↦
```

See Section 7.2.1 [ideal (plural)], page 313; Section 7.3.13 [minres (plural)], page 339; Section 7.2.3 [module (plural)], page 319; Section 7.3.15 [mres (plural)], page 341.

7.3.19 oppose

Syntax: `oppose` (ring_name, name)

Type: poly, vector, ideal, module or matrix (the same type as the second argument)

Purpose: for a given object in the given ring, creates its opposite object in the opposite (Section 7.3.20 [opposite], page 347) ring (the last one is assumed to be the current ring).

Remark: for any object O, $(O^{opp})^{opp} = O$.

```
LIB "ncalg.lib";
def r = makeUsl2();
setring r;
matrix m[3][4];
poly   p  = (h^2-1)*f*e;
vector v  = [1,e*h,0,p];
ideal  i  = h*e, f^2*e,h*f*e;
m         = e,f,h,1,0,h^2, p,0,0,1,e^2,e*f*h+1;
```

```
module mm = module(m);
def b     = opposite(r);
setring b; b;
↦ // coefficients: QQ
↦ // number of vars : 3
↦ //        block   1 : ordering a
↦ //                  : names    H F E
↦ //                  : weights  1 1 1
↦ //        block   2 : ordering ls
↦ //                  : names    H F E
↦ //        block   3 : ordering C
↦ // noncommutative relations:
↦ //    FH=HF-2F
↦ //    EH=HE+2E
↦ //    EF=FE-H
// we will oppose these objects: p,v,i,m,mm
poly P    = oppose(r,p);
vector V  = oppose(r,v);
ideal  I  = oppose(r,i);
matrix M  = oppose(r,m);
module MM = oppose(r,mm);
def c = opposite(b);
setring c;   // now let's check the correctness:
// print compact presentations of objects
print(oppose(b,P)-imap(r,p));
↦ 0
print(oppose(b,V)-imap(r,v));
↦ [0]
print(matrix(oppose(b,I))-imap(r,i));
↦ 0,0,0
print(matrix(oppose(b,M))-imap(r,m));
↦ 0,0,0,0,
↦ 0,0,0,0,
↦ 0,0,0,0
print(matrix(oppose(b,MM))-imap(r,mm));
↦ 0,0,0,0,
↦ 0,0,0,0,
↦ 0,0,0,0
```

See Section 7.3.6 [envelope], page 334; Section 7.3.20 [opposite], page 347.

7.3.20 opposite

Syntax: `opposite` (ring_name)

Type: ring

Purpose: creates an opposite algebra of a given algebra.

Note: activate the ring with the `setring` command.
An opposite algebra of a given algebra (A ,#) is an algebra (A ,*) with the same vector space but with the opposite multiplication, i.e.
$\forall\ f, g \in A^{opp}$, a new multiplication $*$ on A^{opp} is defined to be $f * g := g \sharp f$.

This is an identity functor on commutative algebras.

Remark: Starting from the variables x_1,...,x_N and the ordering `<` of the given algebra, an opposite algebra will have variables X_N,...,X_1 (where the case and the position are reverted). Moreover, it is equipped with an opposed ordering `<_opp` (it is given by the matrix, obtained from the matrix ordering of `<` with the reverse order of columns). Currently not implemented for non-global orderings.

```
LIB "ncalg.lib";
def B = makeQso3(3);
// this algebra is a quantum deformation of U(so_3),
// where the quantum parameter is a 6th root of unity
setring B; B;
↦ // coefficients: QQ[Q]/(Q2-Q+1)
↦ // number of vars : 3
↦ //        block   1 : ordering dp
↦ //                  : names    x y z
↦ //        block   2 : ordering C
↦ // noncommutative relations:
↦ //    yx=(Q-1)*xy+(-Q)*z
↦ //    zx=(-Q)*xz+(-Q+1)*y
↦ //    zy=(Q-1)*yz+(-Q)*x
def Bopp = opposite(B);
setring Bopp;
Bopp;
↦ // coefficients: QQ[Q]/(Q2-Q+1)
↦ // number of vars : 3
↦ //        block   1 : ordering a
↦ //                  : names    Z Y X
↦ //                  : weights  1 1 1
↦ //        block   2 : ordering ls
↦ //                  : names    Z Y X
↦ //        block   3 : ordering C
↦ // noncommutative relations:
↦ //    YZ=(Q-1)*ZY+(-Q)*X
↦ //    XZ=(-Q)*ZX+(-Q+1)*Y
↦ //    XY=(Q-1)*YX+(-Q)*Z
def Bcheck = opposite(Bopp);
setring Bcheck; Bcheck;  // check that (B-opp)-opp = B
↦ // coefficients: QQ[Q]/(Q2-Q+1)
↦ // number of vars : 3
↦ //        block   1 : ordering wp
↦ //                  : names    x y z
↦ //                  : weights  1 1 1
↦ //        block   2 : ordering C
↦ //        block   3 : ordering C
↦ // noncommutative relations:
↦ //    yx=(Q-1)*xy+(-Q)*z
↦ //    zx=(-Q)*xz+(-Q+1)*y
↦ //    zy=(Q-1)*yz+(-Q)*x
```

See Section B.2.6 [Matrix orderings], page 698; Section 7.3.6 [envelope], page 334; Section 7.3.19 [oppose], page 346.

7.3.21 preimage (plural)

Syntax: `preimage` (ring_name, map_name, ideal_name)
`preimage` (ring_name, ideal_expression, ideal_name)

Type: ideal

Purpose: returns the preimage of an ideal under a given map. The second argument has to be a map from the basering to the given ring (or an ideal defining such a map), and the ideal has to be an ideal in the given ring.

Note: To compute the kernel of a map, the preimage of zero has to be determined. Hence there is no special command for computing the kernel of a map in PLURAL.

Remark: In the non-commutative case, it is implemented only for maps A -> B , where A is a commutative ring.

Example:

```
LIB "ncalg.lib";
ring   R = 0,a,dp;
def Usl2 = makeUsl2();
setring Usl2;
poly  C = 4*e*f+h^2-2*h;
// C is a central element of U(sl2)
ideal I = e^3, f^3, h^3-4*h;
ideal Z = 0;  // zero
ideal J = twostd(I); // two-sided GB
ideal K = std(I);    // left GB
map Phi = R,C;  // phi maps a (in R) to C (in U(sl2))
setring R;
ideal PreJ = preimage(Usl2,Phi,J);
// the central character of J
PreJ;
↦ PreJ[1]=a2-8a
factorize(PreJ[1],1);
↦ _[1]=a
↦ _[2]=a-8
// hence, there are two simple characters for J
ideal PreK = preimage(Usl2,Phi,K);
// the central character of K
PreK;
↦ PreK[1]=a3-32a2+192a
factorize(PreK[1],1);
↦ _[1]=a
↦ _[2]=a-24
↦ _[3]=a-8
// hence, there are three simple characters for K
preimage(Usl2, Phi, Z);  // kernel pf phi
↦ _[1]=0
```

See Section 7.2.1 [ideal (plural)], page 313; Section 7.2.2 [map (plural)], page 317; Section 7.2.7 [ring (plural)], page 327.

7.3.22 quotient (plural)

Syntax: `quotient` (ideal_expression, ideal_expression)
`quotient` (module_expression, module_expression)

Type: ideal

Syntax: `quotient` (module_expression, ideal_expression)

Type: module

Purpose: computes the ideal quotient, resp. module quotient. Let `R` be the basering, `I,J` ideals and `M, N` modules in R^n. Then
`quotient(I,J)`$= \{a \in R \mid aJ \subset I\}$,
`quotient(M,J)`$= \{b \in R^n \mid bJ \subset M\}$.

Note: It can only be used for two-sided ideals (bimodules), otherwise the result may have no meaning.

Example:

```
//------ a very simple example ------------
ring r=(0,q),(x,y),Dp;
def R=nc_algebra(q,0); // this algebra is a quantum plane
setring R;
option(returnSB);
poly f1  = x^3+2*x*y^2+2*x^2*y;
poly f2  = y;
poly f3 = x^2;
poly f4 = x+y;
ideal i = f1,f2;
ideal I = twostd(i);
ideal j = f3,f4;
ideal J = twostd(j);
quotient(I,J);
↦ _[1]=y
↦ _[2]=x2
module M = x*freemodule(3), y*freemodule(2);
quotient(M, ideal(x,y));
↦ _[1]=gen(1)
↦ _[2]=gen(2)
↦ _[3]=x*gen(3)
kill r,R;
//------- a bit more involved example
LIB "ncalg.lib";
def Usl2 = makeUsl2();
// this algebra is U(sl_2)
setring Usl2;
ideal i = e3,f3,h3-4*h;
ideal I = std(i);
poly  C = 4*e*f+h^2-2*h;
ideal H = twostd(C-8);
option(returnSB);
ideal Q = quotient(I,H);
// print a compact presentation of Q:
print(matrix(Q));
↦ h,f3,ef2-4f,e2f-6e,e3
```

See Section 7.2.1 [ideal (plural)], page 313; Section 7.2.3 [module (plural)], page 319.

7.3.23 reduce (plural)

Syntax:

`reduce` (poly_expression, ideal_expression)
`reduce` (poly_expression, ideal_expression, int_expression)
`reduce` (vector_expression, ideal_expression)
`reduce` (vector_expression, ideal_expression, int_expression)
`reduce` (vector_expression, module_expression)
`reduce` (vector_expression, module_expression, int_expression)
`reduce` (ideal_expression, ideal_expression)
`reduce` (ideal_expression, ideal_expression, int_expression)
`reduce` (module_expression, ideal_expression)
`reduce` (module_expression, ideal_expression, int_expression)
`reduce` (module_expression, module_expression)
`reduce` (module_expression, module_expression, int_expression)

Type: the type of the first argument

Purpose: reduces a polynomial, vector, ideal or module to its **left** normal form with respect to an ideal or module represented by a left Groebner basis, if the second argument is a left Groebner basis.
returns 0 if and only if the polynomial (resp. vector, ideal, module) is an element (resp. subideal, submodule) of the ideal (resp. module).
Otherwise, the result may have no meaning.
The third (optional) argument 1 of type int forces a reduction which considers only the leading term and does no tail reduction.

Note: The commands `reduce` and `NF` are synonymous.

Example:

```
ring r=(0,a),(e,f,h),Dp;
matrix d[3][3];
d[1,2]=-h;  d[1,3]=2e;  d[2,3]=-2f;
def R=nc_algebra(1,d);
setring R;
// this algebra is U(sl_2) over Q(a)
ideal I = e2, f2, h2-1;
I = std(I);
// print a compact presentation of I
print(matrix(I));
↦ h2-1,fh-f,f2,eh+e,2*ef-h2-h,e2
ideal J = e, h-a;
J = std(J);
// print a compact presentation of J
print(matrix(J));
↦ h+(-a),e
poly z=4*e*f+h^2-2*h;
// z is the central element of U(sl_2)
reduce(z,I);  // the central character of I:
↦ 3
reduce(z,J); // the central character of J:
↦ (a2+2a)
poly nz = z - NF(z,J); // nz will belong to J
reduce(nz,J);
↦ 0
reduce(I,J);
```

```
↦ _[1]=(a2-1)
↦ _[2]=(a-1)*f
↦ _[3]=f2
↦ _[4]=0
↦ _[5]=(-a2+a)
↦ _[6]=0
```

See also Section 7.2.1 [ideal (plural)], page 313; Section 7.2.3 [module (plural)], page 319; Section 7.3.26 [std (plural)], page 355.

7.3.24 ringlist (plural)

Syntax: `ringlist` (ring_expression)
`ringlist` (qring_expression)

Type: list

Purpose: decomposes a ring/qring into a list of 6 (or 4 in the commutative case) components. The first 4 components are common both for the commutative and for the non-commutative cases, the 5th and the 6th appear only in the non-commutative case.

5. upper triangle square matrix with nonzero upper triangle, containing structural coefficients of a G-algebra (this corresponds to the matrix C from the definition of Section 7.4.1 [G-algebras], page 359)
6. square matrix, containing structural polynomials of a G-algebra (this corresponds to the matrix D from the definition of Section 7.4.1 [G-algebras], page 359)

Note: `After modifying a list aquired with ringlist, one can construct a corresponding ring with ring(list).`

Example:

```
// consider the quantized Weyl algebra
ring r = (0,q),(x,d),Dp;
def RS=nc_algebra(q,1);
setring RS; RS;
↦ // coefficients: QQ(q)
↦ // number of vars : 2
↦ //        block   1 : ordering Dp
↦ //                  : names    x d
↦ //        block   2 : ordering C
↦ // noncommutative relations:
↦ //    dx=(q)*xd+1
list l = ringlist(RS);
l;
↦ [1]:
↦    [1]:
↦       0
↦    [2]:
↦       [1]:
↦          q
↦    [3]:
↦       [1]:
↦          [1]:
↦             lp
↦          [2]:
```

```
↦            1
↦    [4]:
↦       _[1]=0
↦ [2]:
↦    [1]:
↦       x
↦    [2]:
↦       d
↦ [3]:
↦    [1]:
↦       [1]:
↦          Dp
↦       [2]:
↦          1,1
↦    [2]:
↦       [1]:
↦          C
↦       [2]:
↦          0
↦ [4]:
↦    _[1]=0
↦ [5]:
↦    _[1,1]=0
↦    _[1,2]=(q)
↦    _[2,1]=0
↦    _[2,2]=0
↦ [6]:
↦    _[1,1]=0
↦    _[1,2]=1
↦    _[2,1]=0
↦    _[2,2]=0
// now, change the relation d*x = q*x*d +1
// into the relation d*x=(q2+1)*x*d + q*d + 1
matrix S = l[5]; // matrix of coefficients
S[1,2] = q^2+1;
l[5] = S;
matrix T = l[6]; // matrix of polynomials
T[1,2] = q*d+1;
l[6] = T;
def rr = ring(l);
setring rr; rr;
↦ // coefficients: QQ(q)
↦ // number of vars : 2
↦ //        block   1 : ordering Dp
↦ //                  : names    x d
↦ //        block   2 : ordering C
↦ // noncommutative relations:
↦ //    dx=(q2+1)*xd+(q)*d+1
```

See also Section 7.2.7 [ring (plural)], page 327; Section 5.1.133 [ringlist], page 251.

7.3.25 slimgb (plural)

Syntax: `slimgb` (ideal_expression)
`slimgb` (module_expression)

Type: same type as argument

Purpose: returns a left Groebner basis of a left ideal or module with respect to the global monomial ordering of the basering.

Note: The commutative algorithm is described in the diploma thesis of Michael Brickenstein "Neue Varianten zur Berechnung von Groebnerbasen", written 2004 under supervision of G.-M. Greuel in Kaiserslautern.

It is designed to keep polynomials or vectors slim (short with small coefficients). Currently best results are examples over function fields (parameters).

The current implementation may not be optimal for weighted degree orderings.

The program only supports the options `prot`, which will give protocol output and `redSB` for returning a reduced Groebner basis. The protocol messages of `slimgb` mean the following:
`M[n,m]` means a parallel reduction of `n` elements with `m` non-zero output elements,
`b` notices an exchange trick described in the thesis and
`e` adds a reductor with non-minimal leading term.

`slimgb` works for grade commutative algebras but not for general GR-algebras. Please use `qslimgb` instead.

For a detailed commutative example see Section A.2.3 [slim Groebner bases], page 642.

Example:

```
LIB "nctools.lib";
LIB "ncalg.lib";
def U = makeUsl(2);
// U is the U(sl_2) algebra
setring U;
ideal I = e^3, f^3, h^3-4*h;
option(redSB);
ideal J = slimgb(I);
J;
↦ J[1]=h3-4h
↦ J[2]=fh2-2fh
↦ J[3]=eh2+2eh
↦ J[4]=2efh-h2-2h
↦ J[5]=f3
↦ J[6]=e3
// compare slimgb with std:
ideal K = std(I);
print(matrix(NF(K,J)));
↦ 0,0,0,0,0,0
print(matrix(NF(J,K)));
↦ 0,0,0,0,0,0
// hence both Groebner bases are equal
;
// another example for exterior algebras
ring r;
def E = Exterior();
setring E; E;
↦ // coefficients: ZZ/32003
```

```
↦ // number of vars : 3
↦ //        block   1 : ordering dp
↦ //                  : names    x y z
↦ //        block   2 : ordering C
↦ // noncommutative relations:
↦ //    yx=-xy
↦ //    zx=-xz
↦ //    zy=-yz
↦ // quotient ring from ideal
↦ _[1]=z2
↦ _[2]=y2
↦ _[3]=x2
slimgb(xy+z);
↦ _[1]=yz
↦ _[2]=xz
↦ _[3]=xy+z
```

See Section 5.1.108 [option], page 230; Section 7.3.26 [std (plural)], page 355.

7.3.26 std (plural)

Syntax: `std` (ideal_expression)
`std` (module_expression)
`std` (ideal_expression`,` poly_expression)
`std` (module_expression`,` vector_expression)

Type: ideal or module

Purpose: returns a left Groebner basis (see Section 7.4.2 [Groebner bases in G-algebras], page 360 for a definition) of an ideal or module with respect to the monomial ordering of the basering.

Use an optional second argument of type poly, resp. vector, to construct the Groebner basis from an already computed one (given as the first argument) and one additional generator (the second argument).

Note: To view the progress of long running computations, use `option(prot)`. (see Section 5.1.108 [option], page 230(prot)).

Example:

```
LIB "ncalg.lib";
option(prot);
def R = makeUsl2();
// this algebra is U(sl_2)
setring R;
ideal I = e2, f2, h2-1;
I=std(I);
↦ 2(2)s
↦ s
↦ s
↦ 3s
↦ (3)2(2)s
↦ s
↦ (4)(3)(2)3s
↦ 2(4)(3)(2)32product criterion:6 chain criterion:3
```

```
I;
↦ I[1]=h2-1
↦ I[2]=fh-f
↦ I[3]=eh+e
↦ I[4]=f2
↦ I[5]=2ef-h-1
↦ I[6]=e2
kill R;
//-------------------------------------------
def RQ = makeQso3(3);
// this algebra is U'_q(so_3),
// where Q is a 6th root of unity
setring RQ;
RQ;
↦ // coefficients: QQ[Q]/(Q2-Q+1)
↦ // number of vars : 3
↦ //        block   1 : ordering dp
↦ //                  : names    x y z
↦ //        block   2 : ordering C
↦ // noncommutative relations:
↦ //    yx=(Q-1)*xy+(-Q)*z
↦ //    zx=(-Q)*xz+(-Q+1)*y
↦ //    zy=(Q-1)*yz+(-Q)*x
ideal J=x2, y2, z2;
J=std(J);
↦ 2(2)s
↦ s
↦ s
↦ 3s
↦ (4)s
↦ 2(3)s
↦ (5)s
↦ (6)s
↦ 1(8)s
↦ (7)(5)s
↦ (3)(2)product criterion:0 chain criterion:17
J;
↦ J[1]=z
↦ J[2]=y
↦ J[3]=x
```

See also Section 7.2.1 [ideal (plural)], page 313; Section 7.2.7 [ring (plural)], page 327.

7.3.27 subst (plural)

Syntax: `subst` (poly_expression,ring_variable, poly_expression)
`subst` (vector _expression,ring_variable, poly_expression)
`subst` (ideal_expression,ring_variable, poly_expression)
`subst` (module _expression,ring_variable, poly_expression)

Type: poly, vector, ideal or module (corresponding to the first argument)

Purpose: substitutes a ring variable by a polynomial.

Example:

```
LIB "ncalg.lib";
def R = makeUsl2();
// this algebra is U(sl_2)
setring R;
poly C  = e*f*h;
poly C1 = subst(C,e,h^3);
C1;
↦ fh4-6fh3+12fh2-8fh
poly C2 = subst(C,f,e+f);
C2;
↦ e2h+efh
```

7.3.28 syz (plural)

Syntax: `syz` (ideal_expression)
`syz` (module_expression)

Type: module

Purpose: computes the first syzygy (i.e., the module of relations of the given generators) of the ideal, resp. module.

Note: if `S` is a matrix of a left syzygy module of left submodule given by matrix `M`, then `transpose(S)*transpose(M) = 0`.

Example:

```
LIB "ncalg.lib";
def R = makeQso3(3);
setring R;
option(redSB);
// we wish to have completely reduced bases:
option(redTail);
ideal tst;
ideal J = x3+x,x*y*z;
print(syz(J));
↦ -yz,
↦ x2+1
ideal K = x+y+z,y+z,z;
module S = syz(K);
print(S);
↦ (Q-1),       (-Q+1)*z,   (-Q)*y,
↦ (Q)*z+(-Q+1),(Q-1)*z+(Q),-x+(Q)*y,
↦ y+(-Q)*z,    x+(-Q),     x+(-Q+1)
tst = ideal(transpose(S)*transpose(K));
// check the property of a syzygy module (tst==0):
size(tst);
↦ 0
 // now compute the Groebner basis of K ...
K = std(K);
// ... print a matrix presentation of K ...
print(matrix(K));
↦ z,y,x
S = syz(K); // ... and its syzygy module
print(S);
↦ y,    x,       (Q-1),
```

```
↦ (Q)*z,(Q),     x,
↦ (Q-1),(-Q+1)*z,(Q)*y
tst = ideal(transpose(S)*transpose(K));
// check the property of a syzygy module (tst==0):
size(tst);
↦ 0
// but the "commutative" syzygy property does not hold
size(ideal(matrix(K)*matrix(S)));
↦ 3
```

See also Section 7.2.1 [ideal (plural)], page 313; Section 7.3.13 [minres (plural)], page 339; Section 7.2.3 [module (plural)], page 319; Section 7.3.15 [mres (plural)], page 341; Section 7.3.18 [nres (plural)], page 345.

7.3.29 twostd

Syntax: `twostd(` ideal_expression);

Type: ideal

Purpose: returns a left Groebner basis of the two-sided ideal, generated by the input, treated as a set of two-sided generators. see Section 5.1.147 [std], page 266

Remark: There are algebras with no two-sided ideals except 0 and the whole algebra (like Weyl algebras).

Example:

```
LIB "ncalg.lib";
def U = makeUsl2(); // this algebra is U(sl_2)
setring U;
ideal i= e^3, f^3, h^3 - 4*h;
option(redSB);
option(redTail);
ideal I = std(i);
print(matrix(I)); // print a compact presentation of I
↦ h3-4h,fh2-2fh,eh2+2eh,2efh-h2-2h,f3,e3
ideal J = twostd(i);
// print a compact presentation of J:
print(matrix(ideal(J[1..6]))); // first 6 gen's
↦ h3-4h,fh2-2fh,eh2+2eh,f2h-2f2,2efh-h2-2h,e2h+2e2
print(matrix(ideal(J[7..size(J)]))); // the rest of gen's
↦ f3,ef2-fh,e2f-eh-2e,e3
// compute the set of elements present in J but not in I
ideal K = NF(J,I);
K = K+0; // simplify K
print(matrix(K));
↦ f2h-2f2,e2h+2e2,ef2-fh,e2f-eh-2e
```

7.3.30 vdim (plural)

Syntax: `vdim (` ideal_expression)
`vdim (` module_expression)

Type: int

Purpose: computes the vector space dimension of the factor-module that equals ring (resp. free module) modulo the ideal (resp. submodule), generated by the leading terms of the given generators.
If the factor-module is not of finite dimension, -1 is returned.

If the generators form a Groebner basis, this is the same as the vector space dimension of the factor-module.

Note: In the non-commutative case, a ring modulo an ideal has a ring stucture if and only if the ideal is two-sided.

Example:

```
ring R=0,(x,y,z),dp;
matrix d[3][3];
d[1,2]=-z;  d[1,3]=2x;  d[2,3]=-2y;
def RS=nc_algebra(1,d); //U(sl_2)
setring RS;
option(redSB); option(redTail);
ideal I=x3,y3,z3-z;
I=std(I);
I;
↦ I[1]=z3-z
↦ I[2]=y3
↦ I[3]=x3
↦ I[4]=y2z2-y2z
↦ I[5]=x2z2+x2z
↦ I[6]=x2y2z-2xyz2-2xyz+2z2+2z
vdim(I);
↦ 21
```

See also Section 7.2.1 [ideal (plural)], page 313; Section 7.3.10 [kbase (plural)], page 337; Section 7.3.26 [std (plural)], page 355.

7.4 Mathematical background (plural)

This section introduces some of the mathematical notions and definitions used throughout the PLURAL manual. For details, please, refer to appropriate articles or text books (see Section 7.4.4 [References (plural)], page 363). A detailed discussion of the subjects in this section can be found in the doctoral thesis [LV] of V. Levandovskyy (see Section 7.4.4 [References (plural)], page 363).

All algebras are assumed to be associative K -algebras for some field K .

7.4.1 G-algebras

Definition (PBW basis)

Let K be a field, and let a K-algebra A be generated by variables $x_1,\ldots,x_n$ subject to some relations. We call A an algebra with **PBW basis** (Poincaré-Birkhoff-Witt basis), if a K–basis of A is $\mathrm{Mon}(x_1,\ldots,x_n)=\{x_1^{a_1}x_2^{a_2}\ldots x_n^{a_n}\mid a_i\in N\cup\{0\}\}$, where a power-product $x_1^{a_1}x_2^{a_2}\ldots x_n^{a_n}$ (in this particular order) is called **a monomial**. For example, x_1x_2 is a monomial, while x_2x_1 is, in general, not a monomial.

Definition (G-algebra)

Let K be a field, and let a K-algebra A be given in terms of generators subject to the following relations:

$A = K\langle x_1, \ldots, x_n \mid \{x_j x_i = c_{ij} \cdot x_i x_j + d_{ij}\}, 1 \leq i < j \leq n\rangle$, where $c_{ij} \in K^*, d_{ij} \in K[x_1, \ldots, x_n]$.

A is called **a G–algebra, if the following conditions hold:**

- there is a monomial well-ordering $<$ on $K[x_1, x_2, \ldots, x_n]$ such that $\forall i < j \ \ \mathrm{LM}(d_{ij}) < x_i x_j$,
- **non-degeneracy conditions**: $\forall \ 1 \leq i < j < k \leq n \ \ : \mathcal{NDC}_{ijk} = 0$, where

$$\mathcal{NDC}_{ijk} = c_{ik}c_{jk} \cdot d_{ij}x_k - x_k d_{ij} + c_{jk} \cdot x_j d_{ik} - c_{ij} \cdot d_{ik}x_j + d_{jk}x_i - c_{ij}c_{ik} \cdot x_i d_{jk}.$$

Note: Note that non-degeneracy conditions simply ensure associativity of multiplication.

Theorem (properties of G-algebras)

Let A be a G-algebra. Then

A has a PBW (Poincaré-Birkhoff-Witt) basis,

A is left and right noetherian,

A is an integral domain.

Setting up a G-algebra

In order to set up a G–algebra one has to do the following steps:

- define a commutative ring $R = K[x_1, \ldots, x_n]$, equipped with a monomial ordering $<$ (see Section 7.2.7.1 [ring declarations (plural)], page 327).
This provides us with the information on a field K (together with its parameters), variables $\{x_i\}$ and an ordering `<`.
From the sequence of variables we will build a G-algebra with the Poincaré-Birkhoff-Witt (PBW) basis $\{x_1^{a_1} x_2^{a_2} \ldots x_n^{a_n}\}$.
- define strictly $n \times n$ upper triangular matrices (of type `matrix`)

1. $C = \{c_{ij}, i < j\}$, with nonzero entries c_{ij} of type number (c_{ij} for $i \geq j$ will be ignored).
2. $D = \{d_{ij}, i < j\}$, with polynomial entries d_{ij} from R (d_{ij} for $i \geq j$ will be ignored).

Call the initialization function `nc_algebra(C,D)` (see Section 7.3.16 [nc_algebra], page 342) with the data C and D.

At present, Plural does not check automatically whether the non-degeneracy conditions hold but it provides a procedure Section 7.7.16.3 [ndcond], page 573 from the library Section 7.7.16 [nctools_lib], page 571 to check this.

7.4.2 Groebner bases in G-algebras

We follow the notations, used in the Singular Manual (e.g. in Section C.1 [Standard bases], page 702).

For a G–algebra A, we denote by ${}_A\langle g_1, \ldots, g_s\rangle$ the left submodule of a free module A^r, generated by elements $\{g_1, \ldots, g_s\} \subset A^r$.

Let $<$ be a fixed monomial well-ordering on the G–algebra A with the Poincaré-Birkhoff-Witt (PBW) basis $\{x^\alpha = x_1^{a_1} x_2^{a_2} \ldots x_n^{a_n}\}$. For a given free module A^r with the basis $\{e_1, \ldots, e_r\}$, $<$ denotes also a fixed module ordering on the set of monomials $\{x^\alpha e_i \mid \alpha \in \mathbf{N}^n, 1 \leq i \leq r\}$.

Definition

For a set $S \subset A^r$, define $L(S)$ to be the K–vector space, spanned on the leading monomials of elements of S, $L(S) = \oplus\{Kx^\alpha e_i \mid \exists s \in S, \mathrm{LM(s)} = x^\alpha e_i\}$. We call $L(S)$ the **span of leading monomials of S.**

Let $I \subset A^r$ be a left A–submodule. A finite set $G \subset I$ is called **a left Groebner basis** of I if and only if $L(G) = L(I)$, that is for any $f \in I \setminus \{0\}$ there exists a $g \in G$ satisfying $\mathrm{LM}(g) \mid \mathrm{LM}(f)$, i.e., if $\mathrm{LM}(f) = x^\alpha e_i$, then $\mathrm{LM}(f) = x^\beta e_i$ with $\beta_j \leq \alpha_j,\ 1 \leq j \leq n.$

Remark: In general non-commutative algorithms are working with well-orderings only (see Section 7.1 [PLURAL], page 312, Section B.2 [Monomial orderings], page 696 and Section 3.3.3 [Term orderings], page 34), unless we deal with grade commutative algebras.

A Groebner basis $G \subset A^r$ is called **minimal** (or **reduced**) if $0 \notin G$ and if $\mathrm{LM}(g) \notin L(G \setminus \{g\})$ for all $g \in G$. Note, that any Groebner basis can be made minimal by deleting successively those g with $\mathrm{LM}(h) \mid \mathrm{LM}(g)$ for some $h \in G \setminus \{g\}$.

For $f \in A^r$ and $G \subset A^r$ we say that f is **completely reduced with respect to** G if no monomial of f is contained in $L(G)$.

Left Normal Form

A map $\mathrm{NF} : A^r \times \{G \mid G \ \text{a (left) Groebner basis}\} \to A^r, (f|G) \mapsto \mathrm{NF}(f|G)$, is called a **(left) normal form** on A^r if for any $f \in A^r$ and any left Groebner basis G the following holds:

(i) $\mathrm{NF}(0|G) = 0$,

(ii) if $\mathrm{NF}(f|G) \neq 0$ then $\mathrm{LM}(g)$ does not divide $\mathrm{LM}(\mathrm{NF}(f|G))$ for all $g \in G$,

(iii) $f - \mathrm{NF}(f|G) \in {}_A\langle G\rangle$.

$\mathrm{NF}(f|G)$ is called a **left normal form of f with respect to** G (note that such a map is not unique).

Remark: As we have already mentioned in the definitions `ideal` and `module` (see Section 7.1 [PLURAL], page 312), Plural works with left normal form only.

Left ideal membership

For a left Groebner basis G of I the following holds: $f \in I$ if and only if the left normal form $\mathrm{NF}(f|G) = 0$.

7.4.3 Syzygies and resolutions (plural)

Syzygies

Let A be a GR-algebra. A **left** (resp. **right**) **syzygy** between k elements $\{f_1, \ldots, f_k\} \subset A^r$ is a k-tuple $(g_1, \ldots, g_k) \in A^k$ satisfying

$$\sum_{i=1}^{k} g_i f_i = 0 \quad \text{resp.} \quad \sum_{i=1}^{k} f_i g_i = 0.$$

The set of all left (resp. right) syzygies between $\{f_1, ..., f_k\}$ is a left (resp. right) submodule S of A^k.

Remark: With respect to the definitions of `ideal` and `module` (see Section 7.1 [PLURAL], page 312), Plural works with left syzygies only (by `syz` we understand a left syzygy). If `S` is a matrix of a left syzygy module of left submodule given by matrix `M`, then `transpose(S)*transpose(M) = 0` (but, in general, $M \cdot S \neq 0$).

Note, that the syzygy modules of I depend on a choice of generators $\{g_1, \ldots, g_s\}$, but one can show that they depend on I uniquely up to direct summands.

Free resolutions

Let $I = {}_A\langle g_1, \ldots, g_s\rangle \subseteq A^r$ and $M = A^r/I$. A **free resolution of** M is a long exact sequence

$$\ldots \longrightarrow F_2 \stackrel{B_2}{\longrightarrow} F_1 \stackrel{B_1}{\longrightarrow} F_0 \longrightarrow M \longrightarrow 0,$$

with `transpose`$(B_{i+1}) \cdot$ `transpose`$(B_i) = 0$

and where the columns of the matrix B_1 generate I . Note, that resolutions over factor-algebras need not to be of finite length.

Generalized Hilbert Syzygy Theorem

For a G–algebra A, generated by n variables, there exists a free resolution of length smaller or equal than n .

Example:

```
ring R=0,(x,y,z),dp;
matrix d[3][3];
d[1,2]=-z;  d[1,3]=2x;  d[2,3]=-2y;
def U=nc_algebra(1,d); // this algebra is U(sl_2)
setring U;
option(redSB); option(redTail);
ideal I=x3,y3,z3-z;
I=std(I);
I;
↦ I[1]=z3-z
↦ I[2]=y3
↦ I[3]=x3
↦ I[4]=y2z2-y2z
↦ I[5]=x2z2+x2z
↦ I[6]=x2y2z-2xyz2-2xyz+2z2+2z
resolution resI = mres(I,0);
resI;
↦  1      5      7      3
↦ U <--  U <--  U <--  U
↦
↦ 0      1      2      3
↦
list l = resI;
// The matrix A_1 is given by
print(matrix(l[1]));
↦ z3-z,y3,x3,y2z2-y2z,x2z2+x2z
// We see that the columns of A_1 generate I.
// The matrix A_2 is given by
print(matrix(l[2]));
```

```
↦ 0,        0,        y2,  x2,  6yz,        -36xy+18z+24,-6xz,
↦ z2+11z+30,0,        0,   0,   2x2z+12x2,  2x3,         0,
↦ 0,        z2-11z+30,0,   0,   0,          -2y3,        2y2z-12y2,
↦ -y,       0,        -z-5,0,   x2y-6xz-30x,9x2,         x3,
↦ 0,        -x,       0,   -z+5,-y3,        -9y2,        -xy2-4yz+28y
ideal tst; // now let us show that the resolution is exact
matrix TST;
TST = transpose(l[3])*transpose(l[2]); // 2nd term
size(ideal(TST));
↦ 0
TST = transpose(l[2])*transpose(l[1]); // 1st term
size(ideal(TST));
↦ 0
```

7.4.4 References (plural)

The Centre for Computer Algebra Kaiserslautern publishes a series of preprints which are electronically available at `http://www.mathematik.uni-kl.de/~zca/Reports_on_ca`. Other sources to check are the following books and articles:

Text books

- [DK] Y. Drozd and V. Kirichenko. Finite dimensional algebras. With an appendix by Vlastimil Dlab. Springer, 1994
- [GPS] Greuel, G.-M. and Pfister, G. with contributions by Bachmann, O. ; Lossen, C. and Schönemann, H. A SINGULAR Introduction to Commutative Algebra. Springer, 2002
- [BGV] Bueso, J.; Gomez Torrecillas, J.; Verschoren, A. Algorithmic methods in non-commutative algebra. Applications to quantum groups. Kluwer Academic Publishers, 2003
- [Kr] Kredel, H. Solvable polynomial rings. Shaker, 1993
- [Li] Huishi Li. Non-commutative Gröbner bases and filtered-graded transfer. Springer, 2002
- [MR] McConnell, J.C. and Robson, J.C. Non-commutative Noetherian rings. With the cooperation of L. W. Small. Graduate Studies in Mathematics. 30. Providence, RI: American Mathematical Society (AMS)., 2001

Descriptions of algorithms and problems

- Havlicek, M. and Klimyk, A. and Posta, S. Central elements of the algebras $U'_q(\mathrm{so}_m)$ and $U'_q(\mathrm{iso}_m)$. arXiv. math. QA/9911130, (1999)
- J. Apel. Gröbnerbasen in nichtkommutativen algebren und ihre anwendung. Dissertation, Universität Leipzig, 1988.
- Apel, J. Computational ideal theory in finitely generated extension rings. Theor. Comput. Sci.(2000), 244(1-2):1-33
- O. Bachmann and H. Schönemann. Monomial operations for computations of Gröbner bases. In Reports On Computer Algebra 18. Centre for Computer Algebra, University of Kaiserslautern (1998)
- D. Decker and D. Eisenbud. Sheaf algorithms using the exterior algebra. In Eisenbud, D.; Grayson, D.; Stillman, M.; Sturmfels, B., editor, Computations in algebraic geometry with Macaulay 2, (2001)

- Jose L. Bueso, J. Gomez Torrecillas and F. J. Lobillo. Computing the Gelfand-Kirillov dimension II. In A. Granja, J. A. Hermida and A. Verschoren eds. Ring Theory and Algebraic Geometry, Lect. Not. in Pure and Appl. Maths., Marcel Dekker, 2001.
- Jose L. Bueso, J. Gomez Torrecillas and F. J. Lobillo. Re-filtering and exactness of the Gelfand-Kirillov dimension. Bulletin des Sciences Mathematiques 125(8), 689-715 (2001).
- J. Gomez Torrecillas and F.J. Lobillo. Global homological dimension of multifiltered rings and quantized enveloping algebras. J. Algebra, 225(2):522-533, 2000.
- N. Iorgov. On the Center of q-Deformed Algebra $U'_q(\mathrm{so}_3)$ Related to Quantum Gravity at q a Root of 1. In Proceedings of IV Int. Conf. "Symmetry in Nonlinear Mathematical Physics",(2001) Kyiv, Ukraine
- A. Kandri-Rody and V. Weispfenning. Non-commutative Gröbner bases in algebras of solvable type. J. Symbolic Computation, 9(1):1-26, 1990.
- Levandovskyy, V. On Gröbner bases for non-commutative G-algebras. In Kredel, H. and Seiler, W.K., editor, Proceedings of the 8th Rhine Workshop on Computer Algebra, 2002.
- [L1] Levandovskyy, V. PBW Bases, Non-degeneracy Conditions and Applications. In Buchweitz, R.-O. and Lenzing, H., editor, Proceedings of the ICRA X conference, Toronto, 2003.
- [LS] Levandovskyy V.; Schönemann, H. Plural - a computer algebra system for noncommutative polynomial algebras. In Proc. of the International Symposium on Symbolic and Algebraic Computation (ISSAC'03). ACM Press, 2003.
- [LV] Levandovskyy, V. Non-commutative Computer Algebra for polynomial algebras: Gröbner bases, applications and implementation. Doctoral Thesis, Universität Kaiserslautern, 2005. Available online at `http://kluedo.ub.uni-kl.de/volltexte/2005/1883/`.
- [L2] Levandovskyy, V. On preimages of ideals in certain non-commutative algebras. In Pfister G., Cojocaru S. and Ufnarovski, V. (editors), Computational Commutative and Non-Commutative Algebraic Geometry, IOS Press, 2005.
- Mora, T. Gröbner bases for non-commutative polynomial rings. Proc. AAECC 3 Lect. N. Comp. Sci, 229: 353-362, 1986.
- Mora, T. An introduction to commutative and non-commutative Groebner bases. Theor. Comp. Sci., 134: 131-173, 1994.
- T. Nüßler and H. Schönemann. Gröbner bases in algebras with zero-divisors. Preprint 244, Universität Kaiserslautern, 1993. Available online at `http://www.mathematik.uni-kl.de/~zca/Reports_on_ca/index.html`.
- Schönemann, H. Singular in a Framework for Polynomial Computations. In Joswig, M. and Takayama, N., editor, Algebra, Geometry and Software Systems, pages 163-176. Springer, 2003.
- T. Yan. The geobucket data structure for polynomials. J. Symbolic Computation, 25(3):285-294, March 1998.

7.5 Graded commutative algebras (SCA)

This section describes basic mathematical notions, definition, and a little bit the implementation of the experimental non-commutative kernel extension Sca of Singular which improves performance of many algorithms in graded commutative algebras.

In order to improve performance of Singular in specific non-commutative algebras one can extend the internal implementation for them in a virtual-method-overloading-like manner. At the moment graded commutative algebras (SCA) and in particular exterior algebras are implemented this way.

Note that grade commutative algebras require no special user actions apart from defining an appropriate non-commutative GR-algebra in SINGULAR. Upon doing that, the supper-commutative structure will be automatically detected and special multiplication will be used. Moreover, in most SCA-aware (e.g. `std`) algorithms special internal improvements will be used (otherwise standard generic non-commutative implementations will be used).

All considered algebras are assumed to be associative K -algebras for some ground field K .

Definition

Polynomial graded commutative algebras are factors of tensor products of commutative algebras with an exterior algebra over a ground field K .

These algebras can be naturally endowed with a Z_2 -grading, where anti-commutative algebra generators have degree 1 and commutative algebra generators (and naturally scalars) have degree 0 . In this particular case they may be considered as super-commutative algebras.

GR-algebra representation

A graded commutative algebra with n commutative and m anti-commutative algebra generators can be represented as factors of the following GR-algebra by some two-sided ideal:

$$K\left\langle x_1,\ldots,x_n;y_1,\ldots,y_m \mid y_j*y_i=-y_iy_j, j>i\right\rangle / \left\langle y_1^2,\ldots,y_m^2\right\rangle .$$

Distinctive features

Graded commutative algebras have zero divisors iff $m>0$: $y_i*y_i=0$.

Unlike other non-commutative algebras one may use any monomial ordering where only the non-commutative variables are required to be global. In particular, commutative variables are allowed to be local. This means that one can work in tensor products of any commutative ring with an exterior algebra.

Example of defining graded commutative algebras in SINGULAR

Given a commutative polynomial ring r , super-commutative structure on it can be introduced as follows:

```
LIB "nctools.lib";
ring r = 0,(a, b, x,y,z, Q, W),(lp(2), dp(3), Dp(2));
// Let us make variables x = var(3), ..., z = var(5) to be anti-commutative
// and add additionally a quotient ideal:
def S = superCommutative(3, 5, ideal(a*W + b*Q*x + z) ); setring S; S;
↦ // coefficients: QQ
↦ // number of vars : 7
↦ //        block   1 : ordering lp
↦ //                  : names    a b
↦ //        block   2 : ordering dp
↦ //                  : names    x y z
↦ //        block   3 : ordering Dp
↦ //                  : names    Q W
↦ //        block   4 : ordering C
↦ // noncommutative relations:
↦ //    yx=-xy
↦ //    zx=-xz
↦ //    zy=-yz
↦ // quotient ring from ideal
↦ _[1]=xz
↦ _[2]=bxyQ-yz
↦ _[3]=aW+bxQ+z
↦ _[4]=z2
```

```
↦ _[5]=y2
↦ _[6]=x2
ideal I = a*x*y + z*Q + b, y*Q + a; I;
↦ I[1]=axy+b+zQ
↦ I[2]=a+yQ
std(I); // Groebner basis is used here since > is global
↦ _[1]=yQW-z
↦ _[2]=yz
↦ _[3]=b+zQ
↦ _[4]=a+yQ
kill r;
// Let's do the same but this time with some local commutative variables:
ring r = 0,(a, b, x,y,z, Q, W),(dp(1), ds(1), lp(3), ds(2));
def S = superCommutative(3, 5, ideal(a*W + b*Q*x + z) ); setring S; S;
↦ // ** redefining S (def S = superCommutative(3, 5, ideal(a*W + b*Q*x + z)\
    ); setring S; S;)
↦ // coefficients: QQ
↦ // number of vars : 7
↦ //        block   1 : ordering lp
↦ //                  : names    a
↦ //        block   2 : ordering ds
↦ //                  : names    b
↦ //        block   3 : ordering lp
↦ //                  : names    x y z
↦ //        block   4 : ordering ds
↦ //                  : names    Q W
↦ //        block   5 : ordering C
↦ // noncommutative relations:
↦ //    yx=-xy
↦ //    zx=-xz
↦ //    zy=-yz
↦ // quotient ring from ideal
↦ _[1]=xz
↦ _[2]=yz-bxyQ
↦ _[3]=aW+z+bxQ
↦ _[4]=x2
↦ _[5]=y2
↦ _[6]=z2
ideal I = a*x*y + z*Q + b, y*Q + a; I;
↦ I[1]=axy+zQ+b
↦ I[2]=a+yQ
std(I);
↦ _[1]=yQW-z-bxQ
↦ _[2]=zQ+b
↦ _[3]=bx
↦ _[4]=by
↦ _[5]=bz
↦ _[6]=b2
↦ _[7]=a+yQ
```

See example of `superCommutative` from the library `nctools.lib`.

Reference: Ph.D thesis by Oleksandr Motsak (2010).

7.6 LETTERPLACE

This section describes mathematical notions and definitions used in the experimental LETTERPLACE extension of SINGULAR.

For further details, please, refer to the papers

[LL09]: Roberto La Scala and Viktor Levandovskyy, "Letterplace ideals and non-commutative Groebner bases", Journal of Symbolic Computation, Volume 44, Issue 10, October 2009, Pages 1374-1393, see `http://dx.doi.org/10.1016/j.jsc.2009.03.002`.

[LL13]: Roberto La Scala and Viktor Levandovskyy, "Skew polynomial rings, Groebner bases and the letterplace embedding of the free associative algebra", Journal of Symbolic Computation, Volume 48, Issue 1, January 2013, Pages 1374-1393, see `http://dx.doi.org/10.1016/j.jsc.2012.05.003` and also `http://arxiv.org/abs/1009.4152`.

All algebras are assumed to be associative K -algebras for some field K .

7.6.1 Free associative algebras

Let V be a K -vector space, spanned by the symbols x_1 ,..., x_n . A free associative algebra in x_1 ,..., x_n over K , denoted by K < x_1 ,..., x_n > is also known as a tensor algebra $T(V)$ of V . It is an infinite dimensional K -vector space, where one can take as a basis the elements of the free monoid < x_1 ,..., x_n >, identifying the identity element (the empty word) with the 1 in K . In other words, the monomials of K < x_1 ,..., x_n > are the words of finite length in the finite alphabet { x_1 ,..., x_n }. The algebra K < x_1 ,..., x_n > is an integral domain, which is not Noetherian for $n > 1$ (hence, a two-sided Groebner basis of a finitely generated ideal might be infinite). The free associative algebra can be regarded as a graded algebra in a natural way.

Any finitely presented associative algebra is isomorphic to a quotient of K < x_1 ,..., x_n > modulo a two-sided ideal.

7.6.2 Groebner bases for two-sided ideals in free associative algebras

We call a total ordering $<$ on the free monoid $X := \langle x_1, \ldots, x_n \rangle$ (where 1 is identified with the identity element) a **monomial ordering if the following conditions hold:**

$<$ is a well-ordering on X, that is $1 < x \ \forall x \in X \setminus \{1\}$,

$\forall p, q, s, t \in X$, if $s < t$, then $p \cdot s \cdot q < p \cdot t \cdot q$,

$\forall p, q, s, t \in X$, if $s = p \cdot t \cdot q$ and $s \neq t$, then $t < s$.

Hence the notions like a leading monomial and a leading coefficient transfer to this situation.

We say that a monomial v divides monomial w , if there exist monomials $p, s \in X$, such that $w = p \cdot v \cdot s$.

In other words v is a proper subword of w .

For a subset $G \subset T := K\langle x_1, \ldots, x_n \rangle$, define a **leading ideal of G to be the two-sided ideal $L(G) = {}_T\langle \ \{lm(g) \mid g \in G \setminus \{0\}\} \ \rangle_T \subseteq T$.**

Let $<$ be a fixed monomial ordering on T. We say that a subset $G \subset I$ is a (two-sided) Groebner basis for the ideal I with respect to $<$, if $L(G) = L(I)$. That is $\forall f \in I \setminus \{0\}$ there exists $g \in G$, such that $lm(g)$ divides $lm(f)$.

7.6.3 Letterplace correspondence

Our work as well as the name letteplace has been inspired by the work of Rota.

Already Feynman and Rota encoded the monomials (words) of the free algebra $x_{i_1} x_{i_2} \ldots x_{i_m} \in K\langle x_1, \ldots, x_n \rangle$ via the double-indexed letterplace (that is encoding the letter (= variable) and its place in the word) monomials $x(i_1|1)x(i_2|2)\ldots x(i_m|m) \in K[X \times N]$, where $X = \{x_1, \ldots, x_n\}$ and N is the monoid of natural numbers, starting with 0 which cannot be used as a place.

Note, that the letterplace algebra $K[X \times N]$ is an infinitely generated commutative polynomial K -algebra. Since $K <$

x_1 ,..., x_n

$>$ is not Noetherian, it is common to perform the computations with modules up to a given degree. In that case the truncated letterplace algebra is finitely generated commutative ring.

In [LL09] a natural shifting on letterplace polynomials was introduced and used. Indeed, there is 1-to-1 correspondence between graded two-sided ideals of a free algebra and so-called letterplace ideals in the letterplace algebra, see [LL09] for details. All the computations take place in the letterplace algebra.

A letterplace monomial of length m is a monomial of a letterplace algebra, such that its m places are exactly 1,2,..., m . In particular, such monomials are multilinear with respect to places. A letterplace polynomial is an element of the K -vector space, spanned by letterplace monomials. A letterplace ideal is generated by letterplace polynomials subject to two kind of operations:

the K-algebra operations of the letterplace algebra and simultaneous shifting of places by any natural number n.

7.6.4 Example of use of Letterplace

The input monomials must be given in a letterplace notation (though we recommend to get used to the letterplace notation, the procedure `freeGBasis` from the Section 7.7.9 [freegb_lib], page 495 provides an alternative).

We recommend first to define a commutative ring $K[X]$ in Singular and equip it with a degree well-ordering. Then, decide what should be the degree bound d and run the procedure `makeLetterplaceRing(d)` from the library Section 7.7.9 [freegb_lib], page 495. This procedure creates a letterplace algebra with an ordering, induced from the given commutative ring $K[X]$. In this algebra, define an ideal `I` as a list of polynomials in the letterplace encoding and run the procedure `letplaceGBasis`. The output is given in the letterplace encoding as well.

Alternatively, one can run the procedure `system("freegb",I,d,n)`, where n is the number of variables of the original commutative ring which does the same computations as `letplaceGBasis`.

It is possible to convert the letterplace presentation of an ideal to a list of strings with the help of procedures `lp2lstr` and `lst2str` from the library Section 7.7.9 [freegb_lib], page 495. This is shown in the second part of the example below.

Yet another anternative is to use the procedure `freeGBasis` from Section 7.7.9 [freegb_lib], page 495 in order to use a different encoding for polynomials in free algebra. No conversion tools are needed in that case.

We illustrate the approach with the following example:

```
LIB "freegb.lib";
ring r = 0,(x,y,z),dp;
int d =4; // degree bound
def R = makeLetterplaceRing(d);
setring R;
```

```
ideal I = x(1)*y(2) + y(1)*z(2), x(1)*x(2) + x(1)*y(2) - y(1)*x(2) - y(1)*y(2);
option(redSB); option(redTail);
ideal J = letplaceGBasis(I);
J;
↦ J[1]=x(1)*y(2)+y(1)*z(2)
↦ J[2]=x(1)*x(2)-y(1)*x(2)-y(1)*y(2)-y(1)*z(2)
↦ J[3]=y(1)*y(2)*y(3)-y(1)*y(2)*z(3)+y(1)*z(2)*y(3)-y(1)*z(2)*z(3)
↦ J[4]=y(1)*y(2)*x(3)+y(1)*y(2)*z(3)+y(1)*z(2)*x(3)+y(1)*z(2)*z(3)
↦ J[5]=y(1)*z(2)*y(3)*y(4)-y(1)*z(2)*y(3)*z(4)+y(1)*z(2)*z(3)*y(4)-y(1)*z(2\
   )*z(3)*z(4)
↦ J[6]=y(1)*z(2)*y(3)*x(4)+y(1)*z(2)*y(3)*z(4)+y(1)*z(2)*z(3)*x(4)+y(1)*z(2\
   )*z(3)*z(4)
↦ J[7]=y(1)*y(2)*z(3)*y(4)-y(1)*y(2)*z(3)*z(4)+y(1)*z(2)*z(3)*y(4)-y(1)*z(2\
   )*z(3)*z(4)
↦ J[8]=y(1)*y(2)*z(3)*x(4)+y(1)*y(2)*z(3)*z(4)+y(1)*z(2)*z(3)*x(4)+y(1)*z(2\
   )*z(3)*z(4)
// ----------------------------------
lp2lstr(J,r); // export an object called @code{@LN} to the ring r
setring r;  // change to the ring r
lst2str(@LN,1); // output the string presentation
↦ [1]:
↦    x*y+y*z
↦ [2]:
↦    x*x-y*x-y*y-y*z
↦ [3]:
↦    y*y*y-y*y*z+y*z*y-y*z*z
↦ [4]:
↦    y*y*x+y*y*z+y*z*x+y*z*z
↦ [5]:
↦    y*z*y*y-y*z*y*z+y*z*z*y-y*z*z*z
↦ [6]:
↦    y*z*y*x+y*z*y*z+y*z*z*x+y*z*z*z
↦ [7]:
↦    y*y*z*y-y*y*z*z+y*z*z*y-y*z*z*z
↦ [8]:
↦    y*y*z*x+y*y*z*z+y*z*z*x+y*z*z*z
```

There are various conversion routines in the library `freegb_lib` (see Section 7.7.9 [freegb_lib], page 495). We work further on implementing more algorithms for non-commutative ideals and modules over free associative algebra.

7.6.5 Release notes of Letterplace

With this functionality it is possible to compute two-sided Groebner basis of a graded two-sided ideal (that is, an ideal, generated by homogeneous polynomials) in a free associative algebra up to a given degree. It is assumed, that each variable has degree 1.

Restrictions of the Letterplace package:

At the moment we provide stable implementation for the homogeneous input only, computations with quasi-homogeneous and general inhomogeneous ideals are under development. (There are no automatic checks.)

Since free algebra is not Noetherian, one has to define an explicit degree bound, up to which a partial Groebner basis will be computed. Note, that `makeLetterplaceRing` procedure stores the bound in the structure of the ring it creates. Thus running `letplaceGBasis` in such a ring does not require the degree bound in its argument.

the options `option(redSB); option(redTail);` must be always activated

we advise to run the computations with the options `option(prot);option(mem);` in order to see the activity journal as well as the memory usage

Further functionality is provided in the libraries for the Letterpace subsystem.

In the Section 7.7.9 [freegb_lib], page 495 one finds e.g. letterplace arithmetics procedures, conversion tools, procedures for creating some common ideals of relations as well as the normal form procedure, providing effective ideal memnership test.

The Section 7.7.7 [fpadim_lib], page 449 contains procedures for computations with vector space basis of a factor algebra including finiteness check and dimension computation.

7.7 Non-commutative libraries

The content of libraries, created for non-commutative extensions of Singular like Plural and Letterplace is described in the following subsections.

Use the `LIB` command for loading of single libraries.

Note: For any computation in Plural, the monomial ordering must be a global ordering.

See also Section D.11.3 [jacobson_lib], page 800 for the diagonalization of matrices over Ore Euclidean domains.

7.7.1 bimodules_lib

Library: bimodules.lib

Purpose: Tools for handling bimodules

Authors: Ann Christina Foldenauer, Christina.Foldenauer@rwth-aachen.de
Viktor Levandovskyy, levandov@math.rwth-aachen.de

Overview:

The main purpose of this library is the handling of bimodules
which will help e.g. to determine weak normal forms of representation matrices
and total divisors within non-commutative, non-simple G-algebras.
We will use modules homomorphisms between a G-algebra and its enveloping algebra
in order to work left Groebner basis theory on bimodules.
Assume we have defined a (non-commutative) G-algebra A over the field K, and an (A,A)-bimodule M.
Instead of working with M over A, we define the enveloping algebra A^{env} = A otimes_K A^{opp}
(this can be done with command envelope(A)) and embed M into A^{env} via imap().
Thus we obtain the left A^{env}-module M otimes 1 in A^{env}.
This has a lot of advantages, because left module theory has much more commands
that are already implemented in SINGULAR:PLURAL. Two important procedures that we can use are std()
which computes the left Groebner basis, and NF() which computes the left normal form.
With the help of this method we are also able to determine the set of bisyzygies of a bimodule.

A built-in command `twostd` in PLURAL computes the two-sided Groebner basis of an ideal
by using the right completion algorithm of [2]. `bistd` from this library uses very different
approach, which is often superior to the right completion.

References:

The procedure bistd() is the implementation of an algorithm M. del Socorro Garcia Roman presented in [1](page 66-78).

[1] Maria del Socorro Garcia Roman, Effective methods in Algebras with PBW bases:
G-algebras and Yang-Baxter Algebras, Ph.D. thesis, Universidad de La Laguna, 2005.
[2] Viktor Levandovskyy, Non-commutative Computer Algebra for polynomial Algebras:
Groebner Bases, Applications and Implementations, Ph.D. thesis, Kaiserlautern, 2005.
[3] N. Jacobson, The theory of rings, AMS, 1943.
[4] P. M. Cohn, Free Rings and their Relations, Academic Press Inc. (London) Ltd., 1971.

Procedures: See also: Section 7.7.12 [ncalg_lib], page 513; Section 7.7.16 [nctools_lib], page 571.

7.7.1.1 bistd

Procedure from library `bimodules.lib` (see Section 7.7.1 [bimodules_lib], page 370).

Usage: bistd(M); M is (two-sided) ideal/module

Return: ideal or module (same type as the argument)

Purpose: Computes the two-sided Groebner basis of an ideal/module with the help the enveloping algebra of the basering, alternative to twostd() for ideals.

Example:

```
LIB "bimodules.lib";
ring w = 0,(x,s),Dp;
def W=nc_algebra(1,s); // 1st shift algebra
setring W;
matrix m[3][3]=[s^2,s+1,0],[s+1,0,s^3-x^2*s],[2*s+1, s^3+s^2, s^2];
print(m);
↦ s2,   s+1,    s+1,
↦ 0,    -x2s+s3,2s+1,
↦ s3+s2,s2,     0
module L = m; module M2 = bistd(L);
print(M2);
↦ 1,1,s+1,
↦ 0,1,0,
↦ 0,0,s2
```

7.7.1.2 bitrinity

Procedure from library `bimodules.lib` (see Section 7.7.1 [bimodules_lib], page 370).

Usage: bitrinity(M); M is (two-sided) ideal/module

Return: ring, the enveloping algebra of the basering, with objects in it. additionally it exports a list L = Coeff, Bisyz.

Theory: Let psi_s be the epimorphism of left R (X) R^{opp} modules:
psi_s(s (X)_K t) = smt := (s_1 m t_1, ... , s_s m t_s) = (\psi(s_1 (X) t_1) , \dots , psi(s_s (X) t_s)) in R^s.
Then psi_s(A) := (psi_s(a_{ij})) for every matrix A in Mat(n x m, R)$.
For a two-sided ideal I = < f_1, ... , f_j> with Groebner basis G = {g_1, ... , g_k} in R, Coeff is the Coefficient-Matrix and BiSyz a bisyzygy matrix.
Let C be the submatrix of Coeff, where C is Coeff without the first row. Then (g_1,...,g_k) = psi_s(C^T * (f_1 ... f_j)^T) and (0,...,0) = psi_s(BiSyz^T * (f_1 ... f_j)^T).
The first row of Coeff (G_1 ... G_n)$ corresponds to the image of the Groebner basis of I: psi_s((G_1 ... G_n)) = G = {g_1 ... g_k }.
For a (R,R)-bimodule M with Groebner basis G = {g_1, ... , g_k} in R^r, Coeff is the coefficient matrix and BiSyz a bisyzygy matrix.
Let C be the submatrix of Coeff, where C is Coeff without the first r rows. Then (g_1 ... g_k) = psi_s(C^T * (f_1 ... f_j)^T) and (0 ... 0) = psi_s(BiSyz^T * (f_1 ... f_j)^T).
The first r rows of Coeff = (G_1 ... G_n) (Here G_i denotes to the i-th column of the first r rows) corresponds to the image of the Groebner basis of M: psi_s((G_1 ... G_n)) = G = {g_1 ... g_k}.

Purpose: This procedure returns a coefficient matrix in the enveloping algebra of the basering R, that gives implicitly the two-sided Groebner basis of a (R,R)-bimodule M and the coefficients that produce the Groebner basis with the help of the originally used generators of M. Additionally it calculates the bisyzygies of M as left-module of the enveloping algebra of R.

Auxiliary procedures:

Note: To get list L = Coeff, BiSyz, we set: def G = bitrinity(); setring G; L; or $L[1]; L[2];.

Example:

```
LIB "bimodules.lib";
ring r = 0,(x,s),dp;
def R = nc_algebra(1,s); setring R; // 1st shift algebra
poly f = x*s + s^2; // only one generator
ideal I = f; // note, two sided Groebner basis of I is xs, s^2
def G = bitrinity(I);
setring G;
print(L[1]); // Coeff
↦ S2, SX,
↦ s-S,-s+S+1
//the first row shows the Groebnerbasis of I consists of
// psi_s(SX) = xs , phi(S^2) = s^2:
// remember phi(a (X) b - c (X) d) = psi_s(a (X) b) - phi(c (X) d) := ab - cd in R.
// psi_s((-s+S+1)*(x*s + s^2)) = psi_s(-xs2-s3+xsS+xs+s2S)
// = -xs^2-s^3+xs^2+xs+s^3 = xs
// psi_s((s-S)*(x*s + s^2)) = psi_s(xs2+s3-xsS-s2S+s2) = s^2
print(L[2]);  //Bisyzygies
↦ sX+sS-2s-SX-S2,x+s-X-S+1,s2-2sS+S2
// e.g. psi_s((x2-2sS+s-X2+2S2+2X+S-1)(x*s + s^2))
// = psi_s(x3s+x2s2-2xs2S+xs2-2s3S+s3-xsX2+2xsS2+2xsX+xsS-xs-s2X2+2s2S2+2s2X-s2S)
// = x^3s+x^2s^2-2xs^3+xs^2-2s^4+s^3-xsx^2+2xs^3+2xsx+xs^2-xs-s^2x^2+2s^4+2s^2x-s^3
// = 0 in R
```

7.7.1.3 liftenvelope

Procedure from library `bimodules.lib` (see Section 7.7.1 [bimodules_lib], page 370).

Usage: liftenvelope(M,g); M ideal/module, g poly

Return: ring, the enveloping algebra of the basering R.
Given a two-sided ideal M in R and a polynomial g in R this procedure returns the enveloping algebra of R. Additionally it exports a list l = C, B; where B is the left Groebner basis of the left-syzygies of M \otimes 1 and C is a vector of coefficients in the enveloping algebra of R such that psi_s(C^T *(f_1 \dots f_n)) = g.
psi_s is an epimorphism of left R (X) R^{opp} modules:
psi_s (s (X)_K t) = smt := (s_1 m t_1, ... , s_s m t_s) = (\psi(s_1 (X) t_1) , \dots , psi(s_s (X) t_s)) in R^s.
Then psi_s(A) := (psi_s(a_{ij})) for every matrix A in Mat(n x m, R)$.

Assume: The second component has to be an element of the first component.

Purpose: This procedure is used for computing total divisors. Let {f_1, ..., f_n} be the generators of the first component and let the second component be called g. Then the returned list l = C, B = (b_1, ..., b_n); defines an affine set A = C + sum_i a_i b_i with (a_1,..,a_n) in the enveloping algebra of the basering R such that psi_s(a^T * (f_1 ... f_n)) = g for all a in A. For certain rings R, we csn find pure tensors within this set A, and if we do, liftenvelope() helps us to decide whether f is a total divisor of g.

Note: To get list l = C, B. we set: def G = liftenvelope(); setring G; l; or l[1]; l[2];.

Example:

```
LIB "bimodules.lib";
ring r = 0,(x,s),dp;
def R = nc_algebra(1,s); setring R;
ideal I = x*s;
poly p = s*x*s*x;  // = (s (x) x) * x*s = (sX) * x*s
p;
↦ x2s2+3xs2+2s2
def J = liftenvelope(I,p);
setring J;
print(l[1]);
↦ 0
//2s+SX = (2s (x) 1) + (1 (x) sx)
print(l[2]);
↦ sX-2s-SX,x-X+1,s2-2sS+S2
// Groebnerbasis of BiSyz(I) as LeftSyz in R^{env}
// We get : 2s+SX + ( sX - 2s -SX) = sX  - a pure tensor!!!!
```

7.7.1.4 CompDecomp

Procedure from library `bimodules.lib` (see Section 7.7.1 [bimodules_lib], page 370).

Usage: CompDecomp(p); p poly

Note: This procedure only works if the basering is an enveloping algebra A^{env} of a (non-commutative) ring A. Thus also the polynomial in the argument has to be in A^{env}.

Return: Returns an ideal I in A^{env}, where the sum of all terms of the argument with the same right side (of the tensor summands) are stored as a generator of I.
Let b != c, then for p = (a (X) b) + (c (X) b) + (a (X) c) the ideal I := CompDecomp(p) is given by: I[1] = (a (X) b) + (c (X) b); I[2] = a (X) c.

Purpose: By decomposing the polynomial we can easily check whether the given polynomial is a pure tensor.

Example:

```
LIB "bimodules.lib";
ring r = 0,(x,s),dp;
def R = nc_algebra(1,s); setring R; //1st shift algebra
def Re = envelope(R); setring Re; //basering is now R^{env} = R (X) R^{opp}
poly f = X*S*x^2+5*x*S*X+S*X; f;
↦ x2SX+x2S+5xSX+SX
ideal I = CompDecomp(f);
print(matrix(I)); // what means that f = (x2+5x+1)*SX + x2*S
↦ x2SX+5xSX+SX,x2S
poly p = x*S+X^2*S+2*s+x*X^2*s+5*x*s; p;
↦ xsX2+5xs+xS+2s+SX2+2SX+S
ideal Q = CompDecomp(p);
print(matrix(Q));
↦ xsX2,5xs+2s,xS+S,SX2,2SX
```

7.7.1.5 isPureTensor

Procedure from library `bimodules.lib` (see Section 7.7.1 [bimodules_lib], page 370).

Usage: isPureTensor(g); g poly

Note: This procedure only works if the basering is an enveloping algebra A^{env} of a (non-commutative) ring A. Thus also the polynomial in the argument has to be in A^{env}.

Return: Returns 0 if g is not a pure tensor and if g is a pure tensor then isPureTensor() returns a vector v with v = a*gen(1)+b*gen(2) = (a,b)^T with a (X) b = g.

Purpose: Checks whether a given polynomial in $\A^{env}$ is a pure tensor. This is also an auxiliary procedure for checking total divisibility.

Example:

```
LIB "bimodules.lib";
ring r = 0,(x,s),dp;
def R = nc_algebra(1,s); setring R; //1st shift algebra
def Re = envelope(R); setring Re; //basering is now R^{env} = R (X) R^{opp}
poly p = x*(x*s)*x + s^2*x; p;
↦ x3s+x2s+xs2+2s2
// p is of the form q(X)1, a pure tensor indeed:
isPureTensor(p);
↦ x3s*gen(1)+x2s*gen(1)+xs2*gen(1)+2s2*gen(1)+gen(2)
// v = transpose( x3s+x2s+xs2+2s2  1 ) i.e. p = x3s+x2s+xs2+2s2 (X) 1
poly g = S*X+ x*s*X+ S^2*x;
g;
↦ xsX+xS2+SX
isPureTensor(g); // indeed g is not a pure tensor
↦ 0
poly d = x*X+s*X+x*S*X+s*S*X;d;
↦ xSX+xX+sSX+sX
isPureTensor(d); // d is a pure tensor indeed
↦ x*gen(1)+s*gen(1)+SX*gen(2)+X*gen(2)
// v = transpose( x+s  S*X+X ) i.e. d = x+s (X) s*x+x
// remember that * denotes to the opposite mulitiplication s*x = xs in R.
```

7.7.1.6 isTwoSidedGB

Procedure from library `bimodules.lib` (see Section 7.7.1 [bimodules_lib], page 370).

Usage: isTwoSidedGB(I); I ideal

Return: Returns 0 if the generators of a given ideal are not two-sided, 1 if they are.\\

Note: This procedure should only be used for non-commutative rings, as every element is two-sided in a commutative ring.

Purpose: Auxiliary procedure for diagonal forms. Let R be a non-commutative ring (e.g. G-algebra), and p in R, this program checks whether p is two-sided i.e. Rp = pR.

Example:

```
LIB "bimodules.lib";
ring r = 0,(x,s),dp;
def R = nc_algebra(1,s); setring R; //1st shift algebra
ideal I = s^2, x*s, s^2 + 3*x*s;
isTwoSidedGB(I); // I is two-sided
↦ 1
ideal J = s^2+x;
isTwoSidedGB(J); // J is not two-sided; twostd(J) = s,x;
↦ 0
```

7.7.2 bfun_lib

Library: bfun.lib

Purpose: Algorithms for b-functions and Bernstein-Sato polynomial

Authors: Daniel Andres, daniel.andres@math.rwth-aachen.de
Viktor Levandovskyy, levandov@math.rwth-aachen.de

Overview: Given a polynomial ring R = K[x_1,...,x_n] and a polynomial F in R, one is interested in the global b-function (also known as Bernstein-Sato polynomial) b(s) in K[s], defined to be the non-zero monic polynomial of minimal degree, satisfying a functional identity L * F^{s+1} = b(s) F^s, for some operator L in D[s] (* stands for the action of differential operator)
By D one denotes the n-th Weyl algebra
K<x_1,...,x_n,d_1,...,d_n | d_j x_j = x_j d_j +1>.
One is interested in the following data:
- Bernstein-Sato polynomial b(s) in K[s],
- the list of its roots, which are known to be rational
- the multiplicities of the roots.

There is a constructive definition of a b-function of a holonomic ideal I in D (that is, an ideal I in a Weyl algebra D, such that D/I is holonomic module) with respect to the given weight vector w: For a polynomial p in D, its initial form w.r.t. (-w,w) is defined as the sum of all terms of p which have maximal weighted total degree where the weight of x_i is -w_i and the weight of d_i is w_i. Let J be the initial ideal of I w.r.t. (-w,w), i.e. the K-vector space generated by all initial forms w.r.t (-w,w) of elements of I. Put s = w_1 x_1 d_1 + ... + w_n x_n d_n. Then the monic generator b_w(s) of the intersection of J with the PID K[s] is called the b-function of I w.r.t. w. Unlike Bernstein-Sato polynomial, general b-function with respect to arbitrary weights need not have rational roots at all. However, b-function of a holonomic ideal is known to be non-zero as well.

References:
[SST] Saito, Sturmfels, Takayama: Groebner Deformations of Hypergeometric Differential Equations (2000),
Noro: An Efficient Modular Algorithm for Computing the Global b-function, (2002).

Procedures: See also: Section 7.7.4 [dmod_lib], page 400; Section 7.7.5 [dmodapp_lib], page 419; Section 7.7.6 [dmodvar_lib], page 445; Section D.6.9 [gmssing_lib], page 778.

7.7.2.1 bfct

Procedure from library `bfun.lib` (see Section 7.7.2 [bfun_lib], page 375).

Usage: bfct(f [,s,t,v]); f a poly, s,t optional ints, v an optional intvec

Return: list of ideal and intvec

Purpose: computes the roots of the Bernstein-Sato polynomial b(s)
for the hypersurface defined by f.

Assume: The basering is commutative and of characteristic 0.

Background:
In this proc, the initial Malgrange ideal is computed according to
the algorithm by Masayuki Noro and then a system of linear equations is
solved by linear reductions.

Note: In the output list, the ideal contains all the roots
and the intvec their multiplicities.
If s<>0, `std` is used for GB computations,
otherwise, and by default, `slimgb` is used.
If t<>0, a matrix ordering is used for Groebner basis computations,
otherwise, and by default, a block ordering is used.
If v is a positive weight vector, v is used for homogenization
computations, otherwise and by default, no weights are used.

Display: If printlevel=1, progress debug messages will be printed,
if printlevel>=2, all the debug messages will be printed.

Example:

```
LIB "bfun.lib";
ring r = 0,(x,y),dp;
poly f = x^2+y^3+x*y^2;
bfct(f);
↦ [1]:
↦    _[1]=-5/6
↦    _[2]=-1
↦    _[3]=-7/6
↦ [2]:
↦    1,1,1
intvec v = 3,2;
bfct(f,1,0,v);
↦ [1]:
↦    _[1]=-5/6
↦    _[2]=-1
```

```
↦    _[3]=-7/6
↦ [2]:
↦    1,1,1
```

7.7.2.2 bfctSyz

Procedure from library `bfun.lib` (see Section 7.7.2 [bfun_lib], page 375).

Usage: bfctSyz(f [,r,s,t,u,v]); f poly, r,s,t,u optional ints, v opt. intvec

Return: list of ideal and intvec

Purpose: computes the roots of the Bernstein-Sato polynomial b(s)
for the hypersurface defined by f

Assume: The basering is commutative and of characteristic 0.

Background:
In this proc, the initial Malgrange ideal is computed according to
the algorithm by Masayuki Noro and then a system of linear equations is
solved by computing syzygies.

Note: In the output list, the ideal contains all the roots and the intvec
their multiplicities.
If r<>0, `std` is used for GB computations in characteristic 0,
otherwise, and by default, `slimgb` is used.
If s<>0, a matrix ordering is used for GB computations, otherwise,
and by default, a block ordering is used.
If t<>0, the computation of the intersection is solely performed over
charasteristic 0, otherwise and by default, a modular method is used.
If u<>0 and by default, `std` is used for GB computations in
characteristic >0, otherwise, `slimgb` is used.
If v is a positive weight vector, v is used for homogenization
computations, otherwise and by default, no weights are used.

Display: If printlevel=1, progress debug messages will be printed,
if printlevel>=2, all the debug messages will be printed.

Example:

```
LIB "bfun.lib";
ring r = 0,(x,y),dp;
poly f = x^2+y^3+x*y^2;
bfctSyz(f);
↦ [1]:
↦    _[1]=-5/6
↦    _[2]=-1
↦    _[3]=-7/6
↦ [2]:
↦    1,1,1
intvec v = 3,2;
bfctSyz(f,0,1,1,0,v);
↦ [1]:
↦    _[1]=-5/6
↦    _[2]=-1
↦    _[3]=-7/6
↦ [2]:
↦    1,1,1
```

7.7.2.3 bfctAnn

Procedure from library `bfun.lib` (see Section 7.7.2 [bfun_lib], page 375).

Usage: bfctAnn(f [,a,b,c]); f a poly, a, b, c optional ints

Return: list of ideal and intvec

Purpose: computes the roots of the Bernstein-Sato polynomial b(s) for the hypersurface defined by f.

Assume: The basering is commutative and of characteristic 0.

Background:
In this proc, Ann(f^s) is computed and then a system of linear equations is solved by linear reductions.

Note: In the output list, the ideal contains all the roots and the intvec their multiplicities.
If a<>0, only f is appended to Ann(f^s), otherwise, and by default, f and all its partial derivatives are appended.
If b<>0, `std` is used for GB computations, otherwise, and by default, `slimgb` is used.
If c<>0, `std` is used for Groebner basis computations of ideals <I+J> when I is already a Groebner basis of <I>.
Otherwise, and by default the engine determined by the switch b is used.
Note that in the case c<>0, the choice for b will be overwritten only for the types of ideals mentioned above.
This means that if b<>0, specifying c has no effect.

Display: If printlevel=1, progress debug messages will be printed,
if printlevel>=2, all the debug messages will be printed.

Example:

```
LIB "bfun.lib";
ring r = 0,(x,y),dp;
poly f = x^2+y^3+x*y^2;
bfctAnn(f);
↦ [1]:
↦    _[1]=-5/6
↦    _[2]=-1
↦    _[3]=-7/6
↦ [2]:
↦    1,1,1
def R = reiffen(4,5); setring R;
RC; // the Reiffen curve in 4,5
↦ xy4+y5+x4
bfctAnn(RC,0,1);
↦ [1]:
↦    _[1]=-9/20
↦    _[2]=-11/20
↦    _[3]=-13/20
↦    _[4]=-7/10
↦    _[5]=-17/20
↦    _[6]=-9/10
↦    _[7]=-19/20
```

```
↦     _[8]=-1
↦     _[9]=-21/20
↦     _[10]=-11/10
↦     _[11]=-23/20
↦     _[12]=-13/10
↦     _[13]=-27/20
↦  [2]:
↦     1,1,1,1,1,1,1,1,1,1,1,1,1
```

7.7.2.4 bfctOneGB

Procedure from library `bfun.lib` (see Section 7.7.2 [bfun_lib], page 375).

Usage: bfctOneGB(f [,s,t]); f a poly, s,t optional ints

Return: list of ideal and intvec

Purpose: computes the roots of the Bernstein-Sato polynomial b(s) for the hypersurface defined by f, using only one GB computation

Assume: The basering is commutative and of characteristic 0.

Background:
In this proc, the initial Malgrange ideal is computed based on the algorithm by Masayuki Noro and combined with an elimination ordering.

Note: In the output list, the ideal contains all the roots and the intvec their multiplicities.
If s<>0, `std` is used for the GB computation, otherwise, and by default, `slimgb` is used.
If t<>0, a matrix ordering is used for GB computations, otherwise, and by default, a block ordering is used.

Display: If printlevel=1, progress debug messages will be printed, if printlevel>=2, all the debug messages will be printed.

Example:

```
LIB "bfun.lib";
ring r = 0,(x,y),dp;
poly f = x^2+y^3+x*y^2;
bfctOneGB(f);
↦  [1]:
↦     _[1]=-5/6
↦     _[2]=-1
↦     _[3]=-7/6
↦  [2]:
↦     1,1,1
bfctOneGB(f,1,1);
↦  [1]:
↦     _[1]=-5/6
↦     _[2]=-1
↦     _[3]=-7/6
↦  [2]:
↦     1,1,1
```

7.7.2.5 bfctIdeal

Procedure from library `bfun.lib` (see Section 7.7.2 [bfun_lib], page 375).

Usage: bfctIdeal(I,w[,s,t]); I an ideal, w an intvec, s,t optional ints

Return: list of ideal and intvec

Purpose: computes the roots of the global b-function of I w.r.t. the weight (-w,w).

Assume: The basering is the n-th Weyl algebra in characteristic 0 and for all 1<=i<=n the identity var(i+n)*var(i)=var(i)*var(i+1)+1 holds, i.e. the sequence of variables is given by x(1),...,x(n),D(1),...,D(n), where D(i) is the differential operator belonging to x(i). Further we assume that I is holonomic.

Background:
In this proc, the initial ideal of I is computed according to the algorithm by Masayuki Noro and then a system of linear equations is solved by linear reductions.

Note: In the output list, say L,
- L[1] of type ideal contains all the rational roots of a b-function,
- L[2] of type intvec contains the multiplicities of above roots,
- optional L[3] of type string is the part of b-function without rational roots.
Note, that a b-function of degree 0 is encoded via L[1][1]=0, L[2]=0 and L[3] is 1 (for nonzero constant) or 0 (for zero b-function).
If s<>0, `std` is used for GB computations in characteristic 0, otherwise, and by default, `slimgb` is used.
If t<>0, a matrix ordering is used for GB computations, otherwise, and by default, a block ordering is used.

Display: If printlevel=1, progress debug messages will be printed, if printlevel>=2, all the debug messages will be printed.

Example:

```
LIB "bfun.lib";
ring @D = 0,(x,y,Dx,Dy),dp;
def D = Weyl();
setring D;
ideal I = 3*x^2*Dy+2*y*Dx,2*x*Dx+3*y*Dy+6; I = std(I);
intvec w1 = 0,1;
intvec w2 = 2,3;
bfctIdeal(I,w1);
↦ [1]:
↦    _[1]=0
↦    _[2]=-2/3
↦    _[3]=-4/3
↦ [2]:
↦    1,1,1
bfctIdeal(I,w2,0,1);
↦ [1]:
↦    _[1]=-6
↦ [2]:
↦    1
ideal J = I[size(I)]; // J is not holonomic by construction
```

```
bfctIdeal(J,w1); //  b-function of D/J w.r.t. w1 is non-zero
↦ WARNING: given ideal is not holonomic
↦ ... setting bound for degree of b-function to 10 and proceeding
↦ [1]:
↦    _[1]=0
↦    _[2]=-2/3
↦    _[3]=-4/3
↦ [2]:
↦    1,1,1
bfctIdeal(J,w2); //  b-function of D/J w.r.t. w2 is zero
↦ WARNING: given ideal is not holonomic
↦ ... setting bound for degree of b-function to 10 and proceeding
↦ // Intersection is zero
↦ [1]:
↦    _[1]=0
↦ [2]:
↦    0
↦ [3]:
↦    0
```

7.7.2.6 pIntersect

Procedure from library `bfun.lib` (see Section 7.7.2 [bfun_lib], page 375).

Usage: pIntersect(f, I [,s]); f a poly, I an ideal, s an optional int

Return: vector, coefficient vector of the monic polynomial

Purpose: compute the intersection of ideal I with the subalgebra K[f]

Assume: I is given as Groebner basis, basering is not a qring.

Note: If the intersection is zero, this proc might not terminate.
If s>0 is given, it is searched for the generator of the intersection
only up to degree s. Otherwise (and by default), no bound is assumed.

Display: If printlevel=1, progress debug messages will be printed,
if printlevel>=2, all the debug messages will be printed.

Example:

```
LIB "bfun.lib";
ring r = 0,(x,y),dp;
poly f = x^2+y^3+x*y^2;
def D = initialMalgrange(f);
setring D;
inF;
↦ inF[1]=x*Dt
↦ inF[2]=2*x*y*Dx+3*y^2*Dx-y^2*Dy-2*x*Dy
↦ inF[3]=2*x^2*Dx+x*y*Dx+x*y*Dy+18*t*Dt+9*x*Dx-x*Dy+6*y*Dy+4*x+18
↦ inF[4]=18*t*Dt^2+6*y*Dt*Dy-y*Dt+27*Dt
↦ inF[5]=y^2*Dt
↦ inF[6]=2*t*y*Dt+2*x*y*Dx+2*y^2*Dx-6*t*Dt-3*x*Dx-x*Dy-2*y*Dy+2*y-6
↦ inF[7]=x*y^2+y^3+x^2
↦ inF[8]=2*y^3*Dx-2*y^3*Dy-3*y^2*Dx-2*x*y*Dy+y^2*Dy-4*y^2+36*t*Dt+18*x*Dx+1\
   2*y*Dy+36
pIntersect(t*Dt,inF);
```

```
↦ gen(4)-1/36*gen(2)
pIntersect(t*Dt,inF,1);
↦ // Try a bound of at least 2
↦ 0
```

7.7.2.7 pIntersectSyz

Procedure from library `bfun.lib` (see Section 7.7.2 [bfun_lib], page 375).

Usage: pIntersectSyz(f, I [,p,s,t]); f poly, I ideal, p,t optial ints, p prime

Return: vector, coefficient vector of the monic polynomial

Purpose: compute the intersection of an ideal I with the subalgebra K[f]

Assume: I is given as Groebner basis.

Note: If the intersection is zero, this procedure might not terminate.
If p>0 is given, this proc computes the generator of the intersection in
char p first and then only searches for a generator of the obtained
degree in the basering. Otherwise, it searches for all degrees by
computing syzygies.
If s<>0, `std` is used for Groebner basis computations in char 0,
otherwise, and by default, `slimgb` is used.
If t<>0 and by default, `std` is used for Groebner basis
computations in char >0, otherwise, `slimgb` is used.

Display: If printlevel=1, progress debug messages will be printed,
if printlevel>=2, all the debug messages will be printed.

Example:

```
LIB "bfun.lib";
ring r = 0,(x,y),dp;
poly f = x^2+y^3+x*y^2;
def D = initialMalgrange(f);
setring D;
inF;
↦ inF[1]=x*Dt
↦ inF[2]=2*x*y*Dx+3*y^2*Dx-y^2*Dy-2*x*Dy
↦ inF[3]=2*x^2*Dx+x*y*Dx+x*y*Dy+18*t*Dt+9*x*Dx-x*Dy+6*y*Dy+4*x+18
↦ inF[4]=18*t*Dt^2+6*y*Dt*Dy-y*Dt+27*Dt
↦ inF[5]=y^2*Dt
↦ inF[6]=2*t*y*Dt+2*x*y*Dx+2*y^2*Dx-6*t*Dt-3*x*Dx-x*Dy-2*y*Dy+2*y-6
↦ inF[7]=x*y^2+y^3+x^2
↦ inF[8]=2*y^3*Dx-2*y^3*Dy-3*y^2*Dx-2*x*y*Dy+y^2*Dy-4*y^2+36*t*Dt+18*x*Dx+1\
   2*y*Dy+36
poly s = t*Dt;
pIntersectSyz(s,inF);
↦ gen(4)-1/36*gen(2)
int p = prime(20000);
pIntersectSyz(s,inF,p,0,0);
↦ gen(4)-1/36*gen(2)
```

7.7.2.8 linReduce

Procedure from library `bfun.lib` (see Section 7.7.2 [bfun_lib], page 375).

Usage: linReduce(f, I [,s,t,u]); f a poly, I an ideal, s,t,u optional ints

Return: poly or list, linear reductum (over field) of f by elements from I

Purpose: reduce a polynomial only by linear reductions (no monomial multiplications)

Note: If s<>0, a list consisting of the reduced polynomial and the coefficient vector of the used reductions is returned, otherwise (and by default) only reduced polynomial is returned.
If t<>0 (and by default) all monomials are reduced (if possible), otherwise, only leading monomials are reduced.
If u<>0 (and by default), the ideal is linearly pre-reduced, i.e. instead of the given ideal, the output of `linReduceIdeal` is used.
If u is set to 0 and the given ideal does not equal the output of `linReduceIdeal`, the result might not be as expected.

Display: If `printlevel`=1, progress debug messages will be printed, if printlevel>=2, all the debug messages will be printed.

Example:

```
LIB "bfun.lib";
ring r = 0,(x,y),dp;
ideal I = 1,y,xy;
poly f = 5xy+7y+3;
poly g = 7x+5y+3;
linReduce(g,I);     // reduces tails
↦ 7x
linReduce(g,I,0,0); // no reductions of tails
↦ 7x+5y+3
linReduce(f,I,1);   // reduces tails and shows reductions used
↦ [1]:
↦    0
↦ [2]:
↦    -5*gen(3)-7*gen(2)-3*gen(1)
f = x3+y2+x2+y+x;
I = x3-y3, y3-x2,x3-y2,x2-y,y2-x;
list l = linReduce(f,I,1);
l;
↦ [1]:
↦    5y
↦ [2]:
↦    gen(5)-4*gen(4)+2*gen(3)-3*gen(2)-3*gen(1)
module M = I;
f - (l[1]-(M*l[2])[1,1]);
↦ 0
```

7.7.2.9 linReduceIdeal

Procedure from library `bfun.lib` (see Section 7.7.2 [bfun_lib], page 375).

Usage: linReduceIdeal(I [,s,t,u]); I an ideal, s,t,u optional ints

Return: ideal or list, linear reductum (over field) of I by its elements

Purpose: reduces a list of polys only by linear reductions (no monomial multiplications)

Note: If s<>0, a list consisting of the reduced ideal and the coefficient vectors of the used reductions given as module is returned.
Otherwise (and by default), only the reduced ideal is returned.
If t<>0 (and by default) all monomials are reduced (if possible), otherwise, only leading monomials are reduced.
If u<>0 (and by default), the ideal is first sorted in increasing order.
If u is set to 0 and the given ideal is not sorted in the way described, the result might not be as expected.

Display: If `printlevel`=1, progress debug messages will be printed, if printlevel>=2, all the debug messages will be printed.

Example:

```
LIB "bfun.lib";
ring r = 0,(x,y),dp;
ideal I = 3,x+9,y4+5x,2y4+7x+2;
linReduceIdeal(I);     // reduces tails
↦ _[1]=0
↦ _[2]=3
↦ _[3]=x
↦ _[4]=y4
linReduceIdeal(I,0,0); // no reductions of tails
↦ _[1]=0
↦ _[2]=3
↦ _[3]=x+9
↦ _[4]=y4+5x
list l = linReduceIdeal(I,1); // reduces tails and shows reductions used
l;
↦ [1]:
↦    _[1]=0
↦    _[2]=3
↦    _[3]=x
↦    _[4]=y4
↦ [2]:
↦    _[1]=gen(4)-2*gen(3)+3*gen(2)-29/3*gen(1)
↦    _[2]=gen(1)
↦    _[3]=gen(2)-3*gen(1)
↦    _[4]=gen(3)-5*gen(2)+15*gen(1)
module M = I;
matrix(l[1]) - M*l[2];
↦ _[1,1]=0
↦ _[1,2]=0
↦ _[1,3]=0
↦ _[1,4]=0
```

7.7.2.10 linSyzSolve

Procedure from library `bfun.lib` (see Section 7.7.2 [bfun_lib], page 375).

Usage: linSyzSolve(I[,s]); I an ideal, s an optional int

Return: vector, coefficient vector of linear combination of 0 in elements of I

Purpose: compute a linear dependency between the elements of an ideal if such one exists

Note: If s<>0, `std` is used for Groebner basis computations, otherwise, `slimgb` is used.
By default, `slimgb` is used in char 0 and `std` in char >0.

Display: If printlevel=1, progress debug messages will be printed, if printlevel>=2, all the debug messages will be printed.

Example:

```
LIB "bfun.lib";
ring r = 0,x,dp;
ideal I = x,2x;
linSyzSolve(I);
↦ gen(2)-2*gen(1)
ideal J = x,x2;
linSyzSolve(J);
↦ 0
```

7.7.2.11 allPositive

Procedure from library `bfun.lib` (see Section 7.7.2 [bfun_lib], page 375).

Usage: allPositive(v); v an intvec

Return: int, 1 if all components of v are positive, or 0 otherwise

Purpose: check whether all components of an intvec are positive

Example:

```
LIB "bfun.lib";
intvec v = 1,2,3;
allPositive(v);
↦ 1
intvec w = 1,-2,3;
allPositive(w);
↦ 0
```

7.7.2.12 scalarProd

Procedure from library `bfun.lib` (see Section 7.7.2 [bfun_lib], page 375).

Usage: scalarProd(v,w); v,w intvecs

Return: int, the standard scalar product of v and w

Purpose: computes the scalar product of two intvecs

Assume: the arguments are of the same size

Example:

```
LIB "bfun.lib";
intvec v = 1,2,3;
intvec w = 4,5,6;
scalarProd(v,w);
↦ 32
```

7.7.2.13 vec2poly

Procedure from library `bfun.lib` (see Section 7.7.2 [bfun_lib], page 375).

Usage: vec2poly(v [,i]); v a vector or an intvec, i an optional int

Return: poly, an univariate polynomial in i-th variable with coefficients given by v

Purpose: constructs an univariate polynomial in K[var(i)] with given coefficients, such that the coefficient at var(i)^{j-1} is v[j].

Note: The optional argument i must be positive, by default i is 1.

Example:

```
LIB "bfun.lib";
ring r = 0,(x,y),dp;
vector v = gen(1) + 3*gen(3) + 22/9*gen(4);
intvec iv = 3,2,1;
vec2poly(v,2);
↦ 22/9y3+3y2+1
vec2poly(iv);
↦ x2+2x+3
```

7.7.3 central_lib

Library: central.lib

Purpose: Computation of central elements of GR-algebras

Author: Oleksandr Motsak, U@D, where U={motsak}, D={mathematik.uni-kl.de}

Overview: A library for computing elements of the center and centralizers of sets of elements in GR-algebras.

Procedures:

7.7.3.1 centralizeSet

Procedure from library `central.lib` (see Section 7.7.3 [central_lib], page 386).

Usage: centralizeSet(F, V); F, V ideals

Input: F, V finite sets of elements of the base algebra

Return: ideal, generated by computed elements

Purpose: computes a vector space basis of the centralizer of the set F in the vector space generated by V over the ground field

Example:

```
LIB "central.lib";
ring A = 0,(e(1..4)),dp;
matrix D[4][4]=0;
D[2,4] = -e(1);
D[3,4] = -e(2);
// This is A_4_1 - the first real Lie algebra of dimension 4.
def A_4_1 = nc_algebra(1,D); setring A_4_1;
ideal F = variablesSorted(); F;
↦ F[1]=e(1)
```

```
↦ F[2]=e(4)
↦ F[3]=e(3)
↦ F[4]=e(2)
// the center of A_4_1 is generated by
// e(1) and -1/2*e(2)^2+e(1)*e(3)
// therefore one may consider computing it in the following way:
// 1. Compute a PBW basis consisting of
//    monomials with exponent <= (1,2,1,0)
ideal V = PBW_maxMonom( e(1) * e(2)^ 2 * e(3) );
// 2. Compute the centralizer of F within the vector space
//    spanned by these monomials:
ideal C = centralizeSet( F, V ); C;
↦ C[1]=e(1)
↦ C[2]=e(2)^2-2*e(1)*e(3)
inCenter(C); // check the result
↦ 1
```

See also: Section 7.7.3.7 [centralizer], page 390; Section 7.7.3.2 [centralizerVS], page 387; Section 7.7.3.11 [inCentralizer], page 393.

7.7.3.2 centralizerVS

Procedure from library `central.lib` (see Section 7.7.3 [central_lib], page 386).

Usage: centralizerVS(F, D); F ideal, D int

Return: ideal, generated by computed elements

Purpose: computes a vector space basis of centralizer(F) up to degree D

Note: D must be non-negative

Example:

```
LIB "central.lib";
ring AA = 0,(x,y,z),dp;
matrix D[3][3]=0;
D[1,2]=-z;  D[1,3]=2*x;  D[2,3]=-2*y;
def A = nc_algebra(1,D); setring A; // this algebra is U(sl_2)
ideal F = x, y;
// find generators of the vector space of elements
// of degree <= 4 commuting with x and y:
ideal C = centralizerVS(F, 4);
C;
↦ C[1]=4xy+z2-2z
↦ C[2]=16x2y2+8xyz2+z4-32xyz-4z3-4z2+16z
inCentralizer(C, F); // check the result
↦ 1
```

See also: Section 7.7.3.4 [centerVS], page 388; Section 7.7.3.7 [centralizer], page 390; Section 7.7.3.11 [inCentralizer], page 393.

7.7.3.3 centralizerRed

Procedure from library `central.lib` (see Section 7.7.3 [central_lib], page 386).

Usage: centralizerRed(F, D[, N]); F ideal, D int, N optional int

Return: ideal, generated by computed elements

Purpose: computes subalgebra generators of centralizer(F) up to degree D.

Note: In general, one cannot compute the whole centralizer(F).
Hence, one has to specify a termination condition via arguments D and/or N.
If D is positive, only centralizing elements up to degree D are computed.
If D is negative, the termination is determined by N only.
If N is given, the computation stops if at least N elements have been found.
Warning: if N is given and bigger than the actual number of generators,
the procedure may not terminate.
Current ordering must be a degree compatible well-ordering.

Example:

```
LIB "central.lib";
ring AA = 0,(x,y,z),dp;
matrix D[3][3]=0;
D[1,2]=-z;  D[1,3]=2*x;  D[2,3]=-2*y;
def A = nc_algebra(1,D); setring A; // this algebra is U(sl_2)
ideal F = x, y;
// find subalgebra generators of degree <= 4 of the subalgebra of
// all elements commuting with x and y:
ideal C = centralizerRed(F, 4);
C;
↦ C[1]=4xy+z2-2z
inCentralizer(C, F); // check the result
↦ 1
```

See also: Section 7.7.3.5 [centerRed], page 389; Section 7.7.3.7 [centralizer], page 390; Section 7.7.3.2 [centralizerVS], page 387; Section 7.7.3.11 [inCentralizer], page 393.

7.7.3.4 centerVS

Procedure from library `central.lib` (see Section 7.7.3 [central_lib], page 386).

Usage: centerVS(D); D int

Return: ideal, generated by computed elements

Purpose: computes a vector space basis of the center up to degree D

Note: D must be non-negative

Example:

```
LIB "central.lib";
ring AA = 0,(x,y,z),dp;
matrix D[3][3]=0;
D[1,2]=-z;  D[1,3]=2*x;  D[2,3]=-2*y;
def A = nc_algebra(1,D); setring A; // this algebra is U(sl_2)
// find a basis of the vector space of all
// central elements of degree <= 4:
ideal Z = centerVS(4);
Z;
↦ Z[1]=4xy+z2-2z
↦ Z[2]=16x2y2+8xyz2+z4-32xyz-4z3-4z2+16z
// note that the second element is the square of the first
// plus a multiple of the first:
Z[2] - Z[1]^2 + 8*Z[1];
```

```
↦ 0
inCenter(Z); // check the result
↦ 1
```

See also: Section 7.7.3.6 [center], page 389; Section 7.7.3.2 [centralizerVS], page 387; Section 7.7.3.10 [inCenter], page 392.

7.7.3.5 centerRed

Procedure from library `central.lib` (see Section 7.7.3 [central_lib], page 386).

Usage: centerRed(D[, N]); D int, N optional int

Return: ideal, generated by computed elements

Purpose: computes subalgebra generators of the center up to degree D

Note: In general, one cannot compute the whole center.
Hence, one has to specify a termination condition via arguments D and/or N.
If D is positive, only central elements up to degree D will be found.
If D is negative, the termination is determined by N only.
If N is given, the computation stops if at least N elements have been found.
Warning: if N is given and bigger than the actual number of generators,
the procedure may not terminate.
Current ordering must be a degree compatible well-ordering.

Example:

```
LIB "central.lib";
ring AA = 0,(x,y,z),dp;
matrix D[3][3]=0;
D[1,2]=z;
def A = nc_algebra(1,D); setring A; // it is a Heisenberg algebra
// find a basis of the vector space of
// central elements of degree <= 3:
ideal VSZ = centerVS(3);
// There should be 3 degrees of z.
VSZ;
↦ VSZ[1]=z
↦ VSZ[2]=z2
↦ VSZ[3]=z3
inCenter(VSZ); // check the result
↦ 1
// find "minimal" central elements of degree <= 3
ideal SAZ = centerRed(3);
// Only 'z' must be computed
SAZ;
↦ SAZ[1]=z
inCenter(SAZ); // check the result
↦ 1
```

See also: Section 7.7.3.6 [center], page 389; Section 7.7.3.4 [centerVS], page 388; Section 7.7.3.3 [centralizerRed], page 387; Section 7.7.3.10 [inCenter], page 392.

7.7.3.6 center

Procedure from library `central.lib` (see Section 7.7.3 [central_lib], page 386).

Usage: center(D[, N]); D int, N optional int

Return: ideal, generated by computed elements

Purpose: computes subalgebra generators of the center up to degree D

Note: In general, one cannot compute the whole center.
Hence, one has to specify a termination condition via arguments D and/or N.
If D is positive, only central elements up to degree D will be found.
If D is negative, the termination is determined by N only.
If N is given, the computation stops if at least N elements have been found.
Warning: if N is given and bigger than the actual number of generators,
the procedure may not terminate.
Current ordering must be a degree compatible well-ordering.

Example:

```
LIB "central.lib";
ring AA = 0,(x,y,z,t),dp;
matrix D[4][4]=0;
D[1,2]=-z;  D[1,3]=2*x;  D[2,3]=-2*y;
def A = nc_algebra(1,D); setring A; // this algebra is U(sl_2) tensored with K[t]
// find generators of the center of degree <= 3:
ideal Z = center(3);
Z;
↦ Z[1]=t
↦ Z[2]=4xy+z2-2z
inCenter(Z); // check the result
↦ 1
// find at least one generator of the center:
ideal Z2 = center(-1, 1);
Z2;
↦ Z2[1]=t
inCenter(Z2); // check the result
↦ 1
```

See also: Section 7.7.3.7 [centralizer], page 390; Section 7.7.3.10 [inCenter], page 392.

7.7.3.7 centralizer

Procedure from library `central.lib` (see Section 7.7.3 [central_lib], page 386).

Usage: centralizer(F, D[, N]); F poly/ideal, D int, N optional int

Return: ideal, generated by computed elements

Purpose: computes subalgebra generators of centralizer(F) up to degree D

Note: In general, one cannot compute the whole centralizer(F).
Hence, one has to specify a termination condition via arguments D and/or N.
If D is positive, only centralizing elements up to degree D will be found.
If D is negative, the termination is determined by N only.
If N is given, the computation stops if at least N elements have been found.
Warning: if N is given and bigger than the actual number of generators,
the procedure may not terminate.
Current ordering must be a degree compatible well-ordering.

Example:

```
LIB "central.lib";
ring AA = 0,(x,y,z),dp;
matrix D[3][3]=0;
D[1,2]=-z; D[1,3]=2*x; D[2,3]=-2*y;
def A = nc_algebra(1,D); setring A; // this algebra is U(sl_2)
poly f = 4*x*y+z^2-2*z; // a central polynomial
f;
↦ 4xy+z2-2z
// find generators of the centralizer of f of degree <= 2:
ideal c = centralizer(f, 2);
c;  // since f is central, the answer consists of generators of A
↦ c[1]=z
↦ c[2]=y
↦ c[3]=x
inCentralizer(c, f); // check the result
↦ 1
// find at least two generators of the centralizer of f:
ideal cc = centralizer(f,-1,2);
cc;
↦ cc[1]=z
↦ cc[2]=y
↦ cc[3]=x
inCentralizer(cc, f); // check the result
↦ 1
poly g = z^2-2*z; // some non-central polynomial
// find generators of the centralizer of g of degree <= 2:
c = centralizer(g, 2);
c;
↦ c[1]=z
↦ c[2]=xy
inCentralizer(c, g); // check the result
↦ 1
// find at least one generator of the centralizer of g:
centralizer(g,-1,1);
↦ _[1]=z
// find at least two generators of the centralizer of g:
cc = centralizer(g,-1,2);
cc;
↦ cc[1]=z
↦ cc[2]=xy
inCentralizer(cc, g); // check the result
↦ 1
```

See also: Section 7.7.3.6 [center], page 389; Section 7.7.3.11 [inCentralizer], page 393.

7.7.3.8 sa_reduce

Procedure from library `central.lib` (see Section 7.7.3 [central_lib], page 386).

Usage: sa_reduce(V); V ideal

Return: ideal, generated by computed elements

Purpose: compute a subalgebra basis of an algebra generated by the elements of V

Note: At the moment the usage of this procedure is limited to G-algebras

Example:

```
LIB "central.lib";
ring AA = 0,(x,y,z),dp;
matrix D[3][3]=0;
D[1,2]=-z; D[1,3]=2*x; D[2,3]=-2*y;
def A = nc_algebra(1,D); setring A; // this algebra is U(sl_2)
poly f = 4*x*y+z^2-2*z; // a central polynomial
ideal I = f, f*f, f*f*f - 10*f*f, f+3*z^3; I;
↦ I[1]=4xy+z2-2z
↦ I[2]=16x2y2+8xyz2+z4-32xyz-4z3+32xy+4z2
↦ I[3]=64x3y3+48x2y2z2+12xyz4+z6-288x2y2z-96xyz3-6z5+352x2y2+224xyz2+2z4-12\
   8xyz+32z3-64xy-40z2
↦ I[4]=3z3+4xy+z2-2z
sa_reduce(I); // should be just f and z^3
↦ _[1]=4xy+z2-2z
↦ _[2]=z3
```

See also: Section 7.7.3.9 [sa_poly_reduce], page 392.

7.7.3.9 sa_poly_reduce

Procedure from library `central.lib` (see Section 7.7.3 [central_lib], page 386).

Usage: sa_poly_reduce(p, V); p poly, V ideal

Return: polynomial, a reduction of p w.r.t. V

Purpose: computes a reduction of the polynomial p w.r.t. the subalgebra generated by elements of V

Note: At the moment the usage of this procedure is limited to G-algebras

Example:

```
LIB "central.lib";
ring AA = 0,(x,y,z),dp;
matrix D[3][3]=0;
D[1,2]=-z; D[1,3]=2*x; D[2,3]=-2*y;
def A = nc_algebra(1,D); setring A; // this algebra is U(sl_2)
poly f = 4*x*y+z^2-2*z; // a central polynomial
sa_poly_reduce(f + 3*f*f + x, ideal(f) ); // should be just 'x'
↦ x
```

See also: Section 7.7.3.8 [sa_reduce], page 391.

7.7.3.10 inCenter

Procedure from library `central.lib` (see Section 7.7.3 [central_lib], page 386).

Usage: inCenter(E); E poly/list/ideal

Return: integer, 1 if E is in the center, 0 otherwise

Purpose: check whether the elements of E are central

Example:

```
LIB "central.lib";
ring R=0,(x,y,z),dp;
matrix D[3][3]=0;
```

```
D[1,2]=-z;
D[1,3]=2*x;
D[2,3]=-2*y;
def r = nc_algebra(1,D); setring r; // this is U(sl_2)
poly p=4*x*y+z^2-2*z;
inCenter(p);
↦ 1
poly f=4*x*y;
inCenter(f);
↦ 0
list l= list( 1, p, p^2, p^3);
inCenter(l);
↦ 1
ideal I= p, f;
inCenter(I);
↦ 0
```

7.7.3.11 inCentralizer

Procedure from library `central.lib` (see Section 7.7.3 [central_lib], page 386).

Usage: inCentralizer(E, S); E poly/list/ideal, S poly/ideal

Return: integer, 1 if E is in the centralizer(S), 0 otherwise

Purpose: check whether the elements of E are in the centralizer(S)

Example:

```
LIB "central.lib";
ring R = 0,(x,y,z),dp;
matrix D[3][3]=0;
D[1,2]=-z;
def r = nc_algebra(1,D); setring r; // the Heisenberg algebra
poly f = x^2;
poly a = z; // 'z' is central => it lies in every centralizer!
poly b = y^2;
inCentralizer(a, f);
↦ 1
inCentralizer(b, f);
↦ 0
list  l = list(1, a);
inCentralizer(l, f);
↦ 1
ideal I = a, b;
inCentralizer(I, f);
↦ 0
printlevel = 2;
inCentralizer(a, f); // yes
↦ 1
inCentralizer(b, f); // no
↦ [1]:
↦    POLY: y2 is NOT in the centralizer of polynomial {x2}
↦ 0
```

7.7.3.12 isCartan

Procedure from library `central.lib` (see Section 7.7.3 [central_lib], page 386).

Usage: isCartan(f); f poly

Purpose: check whether f is a Cartan element.

Return: integer, 1 if f is a Cartan element and 0 otherwise.

Note: f is a Cartan element of the algebra A
if and only if for all g in A there exists C in K such that [f, g] = C * g
if and only if for all variables v_i there exist C in K such that [f, v_i] = C * v_i.

Example:

```
LIB "central.lib";
ring R=0,(x,y,z),dp;
matrix D[3][3]=0;
D[1,2]=-z;
D[1,3]=2*x;
D[2,3]=-2*y;
def r = nc_algebra(1,D); setring r; // this is U(sl_2) with cartan - z
isCartan(z); // yes!
↦ 1
poly p=4*x*y+z^2-2*z;
isCartan(p); // central elements are Cartan elements!
↦ 1
poly f=4*x*y;
isCartan(f); // no way!
↦ 0
isCartan( 10 + p + z ); // scalar + central + cartan
↦ 1
```

7.7.3.13 applyAdF

Procedure from library `central.lib` (see Section 7.7.3 [central_lib], page 386).

Usage: applyAdF(B, f); B ideal, f poly

Purpose: Apply Ad_f to every element of B

Return: ideal, generated by Ad_f(B[i]), 1<=i<=size(B)

Note: Ad_f(v) := [f, v] = f*v - v*f

Example:

```
LIB "central.lib";
ring AA = 0,(e,f,h),dp;
matrix D[3][3]=0;
D[1,2]=-h;  D[1,3]=2*e;  D[2,3]=-2*f;
def A = nc_algebra(1,D); setring A; // this algebra is U(sl_2)
// Let us consider the linear map Ad_{e} from A_2 into A.
// Compute the PBW basis of A_2:
ideal Basis = PBW_maxDeg( 2 ); Basis;
↦ Basis[1]=e
↦ Basis[2]=f
↦ Basis[3]=h
↦ Basis[4]=h2
```

```
↦ Basis[5]=fh
↦ Basis[6]=f2
↦ Basis[7]=eh
↦ Basis[8]=ef
↦ Basis[9]=e2
// Compute images of basis elements under the linear map Ad_e:
ideal Image = applyAdF( Basis, e ); Image;
↦ Image[1]=0
↦ Image[2]=h
↦ Image[3]=-2e
↦ Image[4]=-4eh-4e
↦ Image[5]=-2ef+h2+2h
↦ Image[6]=2fh-2f
↦ Image[7]=-2e2
↦ Image[8]=eh
↦ Image[9]=0
// Now we have a linear map given by: Basis_i --> Image_i
// Let's compute its kernel K:
// 1. compute syzygy module C:
module C = linearMapKernel( Image ); C;
↦ C[1]=gen(1)
↦ C[2]=gen(8)+1/4*gen(4)-1/2*gen(3)
↦ C[3]=gen(9)
// 2. compute corresponding combinations of basis vectors:
ideal K = linearCombinations(Basis, C); K;
↦ K[1]=e
↦ K[2]=ef+1/4h2-1/2h
↦ K[3]=e2
// Let's check that Ad_e(K) is zero:
applyAdF( K, e );
↦ _[1]=0
↦ _[2]=0
↦ _[3]=0
```

See also: Section 7.7.3.14 [linearMapKernel], page 395.

7.7.3.14 linearMapKernel

Procedure from library `central.lib` (see Section 7.7.3 [central_lib], page 386).

Usage: linearMapKernel(Images); Images ideal

Purpose: Computes the syzygy module of the linear map given by Images.

Return: syzygy module, or int(0) if all images are zeroes

Example:

```
LIB "central.lib";
ring AA = 0,(e,f,h),dp;
matrix D[3][3]=0;
D[1,2]=-h;  D[1,3]=2*e;  D[2,3]=-2*f;
def A = nc_algebra(1,D); // this algebra is U(sl_2)
setring A;
// Let us consider the linear map Ad_{e} from A_2 into A.
// Compute the PBW basis of A_2:
ideal Basis = PBW_maxDeg( 2 ); Basis;
```

```
↦ Basis[1]=e
↦ Basis[2]=f
↦ Basis[3]=h
↦ Basis[4]=h2
↦ Basis[5]=fh
↦ Basis[6]=f2
↦ Basis[7]=eh
↦ Basis[8]=ef
↦ Basis[9]=e2
// Compute images of basis elements under the linear map Ad_e:
ideal Image = applyAdF( Basis, e ); Image;
↦ Image[1]=0
↦ Image[2]=h
↦ Image[3]=-2e
↦ Image[4]=-4eh-4e
↦ Image[5]=-2ef+h2+2h
↦ Image[6]=2fh-2f
↦ Image[7]=-2e2
↦ Image[8]=eh
↦ Image[9]=0
// Now we have a linear map given by: Basis_i --> Image_i
// Let's compute its kernel K:
// 1. compute syzygy module C:
module C = linearMapKernel( Image ); C;
↦ C[1]=gen(1)
↦ C[2]=gen(8)+1/4*gen(4)-1/2*gen(3)
↦ C[3]=gen(9)
// 2. compute corresponding combinations of basis vectors:
ideal K = linearCombinations(Basis, C); K;
↦ K[1]=e
↦ K[2]=ef+1/4h2-1/2h
↦ K[3]=e2
// Let's check that Ad_e(K) is zero:
ideal Z = applyAdF( K, e ); Z;
↦ Z[1]=0
↦ Z[2]=0
↦ Z[3]=0
// Now linearMapKernel will return a single integer 0:
def CC  = linearMapKernel(Z); typeof(CC); CC;
↦ int
↦ 0
```

See also: Section 7.7.3.13 [applyAdF], page 394; Section 7.7.3.14 [linearMapKernel], page 395.

7.7.3.15 linearCombinations

Procedure from library `central.lib` (see Section 7.7.3 [central_lib], page 386).

Usage: linearCombinations(Basis, C); Basis ideal, C module

Purpose: forms linear combinations of elements from Basis by replacing gen(i) by Basis[i] in C

Return: ideal generated by computed linear combinations

Example:

```
LIB "central.lib";
ring AA = 0,(e,f,h),dp;
matrix D[3][3]=0;
D[1,2]=-h;  D[1,3]=2*e;  D[2,3]=-2*f;
def A = nc_algebra(1,D); setring A; // this algebra is U(sl_2)
// Let us consider the linear map Ad_{e} from A_2 into A.
// Compute the PBW basis of A_2:
ideal Basis = PBW_maxDeg( 2 ); Basis;
↦ Basis[1]=e
↦ Basis[2]=f
↦ Basis[3]=h
↦ Basis[4]=h2
↦ Basis[5]=fh
↦ Basis[6]=f2
↦ Basis[7]=eh
↦ Basis[8]=ef
↦ Basis[9]=e2
// Compute images of basis elements under the linear map Ad_e:
ideal Image = applyAdF( Basis, e ); Image;
↦ Image[1]=0
↦ Image[2]=h
↦ Image[3]=-2e
↦ Image[4]=-4eh-4e
↦ Image[5]=-2ef+h2+2h
↦ Image[6]=2fh-2f
↦ Image[7]=-2e2
↦ Image[8]=eh
↦ Image[9]=0
// Now we have a linear map given by: Basis_i --> Image_i
// Let's compute its kernel K:
// 1. compute syzygy module C:
module C = linearMapKernel( Image ); C;
↦ C[1]=gen(1)
↦ C[2]=gen(8)+1/4*gen(4)-1/2*gen(3)
↦ C[3]=gen(9)
// 2. compute corresponding combinations of basis vectors:
ideal K = linearCombinations(Basis, C); K;
↦ K[1]=e
↦ K[2]=ef+1/4h2-1/2h
↦ K[3]=e2
// Let's check that Ad_e(K) is zero:
applyAdF( K, e );
↦ _[1]=0
↦ _[2]=0
↦ _[3]=0
```

See also: Section 7.7.3.13 [applyAdF], page 394; Section 7.7.3.14 [linearMapKernel], page 395.

7.7.3.16 variablesStandard

Procedure from library `central.lib` (see Section 7.7.3 [central_lib], page 386).

Usage: variablesStandard();

Return: ideal, generated by algebra variables

Purpose: computes the set of algebra variables taken in their natural order

Example:

```
LIB "central.lib";
ring AA = 0,(x,y,z),dp;
matrix D[3][3]=0;
D[1,2]=-z;  D[1,3]=2*x;  D[2,3]=-2*y;
def A = nc_algebra(1,D); setring A; // this algebra is U(sl_2)
// Variables in their natural order:
variablesStandard();
↦ _[1]=x
↦ _[2]=y
↦ _[3]=z
```

See also: Section 7.7.3.17 [variablesSorted], page 398.

7.7.3.17 variablesSorted

Procedure from library `central.lib` (see Section 7.7.3 [central_lib], page 386).

Usage: variablesSorted();

Return: ideal, generated by sorted algebra variables

Purpose: computes the set of algebra variables sorted so that
Cartan variables go first

Note: This is a heuristics for the computation of the center:
it is better to compute centralizers of Cartan variables first since in this
case we can omit solving the system of equations.

Example:

```
LIB "central.lib";
ring AA = 0,(x,y,z),dp;
matrix D[3][3]=0;
D[1,2]=-z;  D[1,3]=2*x;  D[2,3]=-2*y;
def A = nc_algebra(1,D); setring A; // this algebra is U(sl_2)
// There is only one Cartan variable - z in U(sl_2),
// it must go 1st:
variablesSorted();
↦ _[1]=z
↦ _[2]=y
↦ _[3]=x
```

See also: Section 7.7.3.16 [variablesStandard], page 397.

7.7.3.18 PBW_eqDeg

Procedure from library `central.lib` (see Section 7.7.3 [central_lib], page 386).

Usage: PBW_eqDeg(Deg); Deg int

Purpose: Compute PBW elements of a given degree.

Return: ideal consisting of found elements.

Note: Unit is omitted. Weights are ignored!

Example:

```
LIB "central.lib";
ring AA = 0,(e,f,h),dp;
matrix D[3][3]=0;
D[1,2]=-h;  D[1,3]=2*e;  D[2,3]=-2*f;
def A = nc_algebra(1,D); setring A; // this algebra is U(sl_2)
// PBW Basis of A_2 \ A_1 - monomials of degree == 2:
PBW_eqDeg( 2 );
↦ _[1]=h2
↦ _[2]=fh
↦ _[3]=f2
↦ _[4]=eh
↦ _[5]=ef
↦ _[6]=e2
```

7.7.3.19 PBW_maxDeg

Procedure from library `central.lib` (see Section 7.7.3 [central_lib], page 386).

Usage: PBW_maxDeg(MaxDeg); MaxDeg int

Purpose: Compute PBW elements up to a given maximal degree.

Return: ideal consisting of found elements.

Note: unit is omitted. Weights are ignored!

Example:

```
LIB "central.lib";
ring AA = 0,(e,f,h),dp;
matrix D[3][3]=0;
D[1,2]=-h;  D[1,3]=2*e;  D[2,3]=-2*f;
def A = nc_algebra(1,D); // this algebra is U(sl_2)
setring A;
// PBW Basis of A_2 - monomials of degree <= 2, without unit:
PBW_maxDeg( 2 );
↦ _[1]=e
↦ _[2]=f
↦ _[3]=h
↦ _[4]=h2
↦ _[5]=fh
↦ _[6]=f2
↦ _[7]=eh
↦ _[8]=ef
↦ _[9]=e2
```

7.7.3.20 PBW_maxMonom

Procedure from library `central.lib` (see Section 7.7.3 [central_lib], page 386).

Usage: PBW_maxMonom(m); m poly

Purpose: Compute PBW elements up to a given maximal one.

Return: ideal consisting of found elements.

Note: Unit is omitted. Weights are ignored!

Example:

```
LIB "central.lib";
ring AA = 0,(e,f,h),dp;
matrix D[3][3]=0;
D[1,2]=-h;  D[1,3]=2*e;  D[2,3]=-2*f;
def A = nc_algebra(1,D); // this algebra is U(sl_2)
setring A;
// At most 1st degree in e, h and at most 2nd degree in f, unit is omitted:
PBW_maxMonom( e*(f^2)* h );
↦ _[1]=e
↦ _[2]=f
↦ _[3]=ef
↦ _[4]=f2
↦ _[5]=ef2
↦ _[6]=h
↦ _[7]=eh
↦ _[8]=fh
↦ _[9]=efh
↦ _[10]=f2h
↦ _[11]=ef2h
```

7.7.4 dmod_lib

Library: dmod.lib

Purpose: Algorithms for algebraic D-modules

Authors: Viktor Levandovskyy, levandov@math.rwth-aachen.de
Jorge Martin Morales, jorge@unizar.es

Overview: Let K be a field of characteristic 0. Given a polynomial ring
R = K[x_1,...,x_n] and a polynomial F in R,
one is interested in the R[1/F]-module of rank one, generated by
the symbol F^s for a symbolic discrete variable s.
In fact, the module R[1/F]*F^s has a structure of a D(R)[s]-module, where D(R)
is an n-th Weyl algebra K<x_1,...,x_n,d_1,...,d_n | d_j x_j = x_j d_j +1> and
D(R)[s] = D(R) tensored with K[s] over K.
Constructively, one needs to find a left ideal I = I(F^s) in D(R), such
that K[x_1,...,x_n,1/F]*F^s is isomorphic to D(R)/I as a D(R)-module.
We often write just D for D(R) and D[s] for D(R)[s].
One is interested in the following data:
- Ann F^s = I = I(F^s) in D(R)[s], denoted by LD in the output
- global Bernstein polynomial in K[s], denoted by bs,
- its minimal integer root s0, the list of all roots of bs, which are known
to be rational, with their multiplicities, which is denoted by BS
- Ann F^s0 = I(F^s0) in D(R), denoted by LD0 in the output
(LD0 is a holonomic ideal in D(R))
- Ann^(1) F^s in D(R)[s], denoted by LD1 (logarithmic derivations)
- an operator in D(R)[s], denoted by PS, such that the functional equality
PS*F^(s+1) = bs*F^s holds in K[x_1,...,x_n,1/F]*F^s.

References:

We provide the following implementations of algorithms:
(OT) the classical Ann F^s algorithm from Oaku and Takayama (Journal of

Pure and Applied Math., 1999),
(LOT) Levandovskyy's modification of the Oaku-Takayama algorithm (ISSAC 2007)
(BM) the Ann F^s algorithm by Briancon and Maisonobe (Remarques sur
l'ideal de Bernstein associe a des polynomes, preprint, 2002)
(LM08) V. Levandovskyy and J. Martin-Morales, ISSAC 2008
(C) Countinho, A Primer of Algebraic D-Modules,
(SST) Saito, Sturmfels, Takayama 'Groebner Deformations of Hypergeometric
Differential Equations', Springer, 2000

Guide:
- Ann F^s = I(F^s) = LD in D(R)[s] can be computed by Sannfs [BM, OT, LOT]
- Ann^(1) F^s in D(R)[s] can be computed by Sannfslog
- global Bernstein polynomial bs in K[s] can be computed by bernsteinBM
- Ann F^s0 = I(F^s0) = LD0 in D(R) can be computed by annfs0, annfs, annfsBM,
annfsOT, annfsLOT, annfs2, annfsRB etc.
- all the relevant data to F^s (LD, LD0, bs, PS) are computed by operatorBM
- operator PS can be computed via operatorModulo or operatorBM

- annihilator of F^{s1} for a number s1 is computed with annfspecial
- annihilator of F_1^s_1 * ... * F_p^s_p is computed with annfsBMI
- computing the multiplicity of a rational number r in the Bernstein poly
of a given ideal goes with checkRoot
- check, whether a given univariate polynomial divides the Bernstein poly
goes with checkFactor

Procedures: See also: Section 7.7.2 [bfun_lib], page 375; Section 7.7.5 [dmodapp_lib], page 419; Section 7.7.6 [dmodvar_lib], page 445; Section D.6.9 [gmssing_lib], page 778.

7.7.4.1 annfs

Procedure from library `dmod.lib` (see Section 7.7.4 [dmod_lib], page 400).

Usage: annfs(f [,S,eng]); f a poly, S a string, eng an optional int

Return: ring

Purpose: compute the D-module structure of basering[1/f]*f^s with the algorithm
given in S and with the Groebner basis engine given in "eng"

Note: activate the output ring with the `setring` command.
String S; S can be one of the following:
'bm' (default) - for the algorithm of Briancon and Maisonobe,
'ot' - for the algorithm of Oaku and Takayama,
'lot' - for the Levandovskyy's modification of the algorithm of OT.
If eng `<>`0, `std` is used for Groebner basis computations,
otherwise and by default `slimgb` is used.
In the output ring:
- the ideal LD (which is a Groebner basis) is the needed D-module structure,
- the list BS contains roots and multiplicities of a BS-polynomial of f.

Display: If `printlevel`=1, progress debug messages will be printed,
if `printlevel>`=2, all the debug messages will be printed.

Example:

```
LIB "dmod.lib";
ring r = 0,(x,y,z),Dp;
poly F = z*x^2+y^3;
def A  = annfs(F); // here, the default BM algorithm will be used
setring A; // the Weyl algebra in (x,y,z,Dx,Dy,Dz)
LD; //the annihilator of F^{-1} over A
↦ LD[1]=y*Dy+3*z*Dz+3
↦ LD[2]=x*Dx-2*z*Dz
↦ LD[3]=x^2*Dy-3*y^2*Dz
↦ LD[4]=3*y^2*Dx-2*x*z*Dy
↦ LD[5]=y^3*Dz+x^2*z*Dz+x^2
↦ LD[6]=2*x*z*Dy^2+9*y*z*Dx*Dz+3*y*Dx
↦ LD[7]=9*y*z*Dx^2*Dz+4*z^2*Dy^2*Dz+3*y*Dx^2+2*z*Dy^2
↦ LD[8]=4*z^2*Dy^3*Dz-27*z^2*Dx^2*Dz^2+2*z*Dy^3-54*z*Dx^2*Dz-6*Dx^2
BS; // roots with multiplicities of BS polynomial
↦ [1]:
↦    _[1]=-7/6
↦    _[2]=-5/3
↦    _[3]=-4/3
↦    _[4]=-5/6
↦    _[5]=-1
↦ [2]:
↦    1,1,1,1,1
```

7.7.4.2 annfspecial

Procedure from library `dmod.lib` (see Section 7.7.4 [dmod_lib], page 400).

Usage: annfspecial(I,F,mir,n); I an ideal, F a poly, int mir, number n

Return: ideal

Purpose: compute the annihilator ideal of F^n in the Weyl Algebra
for the given rational number n

Assume: The basering is D[s] and contains 's' explicitly as a variable,
the ideal I is the Ann F^s in D[s] (obtained with e.g. SannfsBM),
the integer 'mir' is the minimal integer root of the BS polynomial of F,
and the number n is rational.

Note: We compute the real annihilator for any rational value of n (both
generic and exceptional). The implementation goes along the lines of
the Algorithm 5.3.15 from Saito-Sturmfels-Takayama.

Display: If printlevel=1, progress debug messages will be printed,
if printlevel>=2, all the debug messages will be printed.

Example:

```
LIB "dmod.lib";
ring r = 0,(x,y),dp;
poly F = x3-y2;
def  B = annfs(F);  setring B;
minIntRoot(BS[1],0);
↦ -1
// So, the minimal integer root is -1
setring r;
```

```
def  A = SannfsBM(F);
setring A;
poly F = x3-y2;
annfspecial(LD,F,-1,3/4); // generic root
↦ _[1]=4*x*Dx+6*y*Dy-9
↦ _[2]=3*x^2*Dy+2*y*Dx
↦ _[3]=18*x*y*Dy^2-8*y*Dx^2-33*x*Dy
↦ _[4]=54*y^2*Dy^3+16*y*Dx^3+66*x*Dx*Dy-9*y*Dy^2+66*Dy
annfspecial(LD,F,-1,-2);  // integer but still generic root
↦ _[1]=2*x*Dx+3*y*Dy+12
↦ _[2]=3*x^2*Dy+2*y*Dx
↦ _[3]=9*x*y*Dy^2-4*y*Dx^2+33*x*Dy
↦ _[4]=27*y^2*Dy^3+8*y*Dx^3-66*x*Dx*Dy+144*y*Dy^2-66*Dy
annfspecial(LD,F,-1,1);   // exceptional root
↦ _[1]=Dx*Dy
↦ _[2]=2*x*Dx+3*y*Dy-6
↦ _[3]=Dy^3
↦ _[4]=y*Dy^2-Dy
↦ _[5]=3*x*Dy^2+Dx^2
↦ _[6]=3*x^2*Dy+2*y*Dx
↦ _[7]=Dx^3+3*Dy^2
↦ _[8]=y*Dx^2+3*x*Dy
```

7.7.4.3 Sannfs

Procedure from library `dmod.lib` (see Section 7.7.4 [dmod_lib], page 400).

Usage: Sannfs(f [,S,eng]); f a poly, S a string, eng an optional int

Return: ring

Purpose: compute the D-module structure of basering[f^s] with the algorithm given in S and with the Groebner basis engine given in eng

Note: activate the output ring with the `setring` command.
The value of a string S can be
'bm' (default) - for the algorithm of Briancon and Maisonobe,
'lot' - for the Levandovskyy's modification of the algorithm of OT,
'ot' - for the algorithm of Oaku and Takayama.
If eng `<>`0, `std` is used for Groebner basis computations,
otherwise, and by default `slimgb` is used.
In the output ring:
- the ideal LD is the needed D-module structure.

Display: If `printlevel`=1, progress debug messages will be printed,
if `printlevel`>=2, all the debug messages will be printed.

Example:

```
LIB "dmod.lib";
ring r = 0,(x,y,z),Dp;
poly F = x^3+y^3+z^3;
printlevel = 0;
def A  = Sannfs(F); // here, the default BM algorithm will be used
setring A;
LD;
```

```
↦ LD[1]=z^2*Dy-y^2*Dz
↦ LD[2]=x*Dx+y*Dy+z*Dz-3*s
↦ LD[3]=z^2*Dx-x^2*Dz
↦ LD[4]=y^2*Dx-x^2*Dy
```

7.7.4.4 Sannfslog

Procedure from library `dmod.lib` (see Section 7.7.4 [dmod_lib], page 400).

Usage: Sannfslog(f [,eng]); f a poly, eng an optional int

Return: ring

Purpose: compute the D-module structure of basering[1/f]*f^s

Note: activate the output ring with the `setring` command.
In the output ring D[s], the ideal LD1 is generated by the elements
in Ann F^s in D[s], coming from logarithmic derivations.
If eng <>0, `std` is used for Groebner basis computations,
otherwise, and by default `slimgb` is used.

Display: If `printlevel`=1, progress debug messages will be printed,
if `printlevel`>=2, all the debug messages will be printed.

Example:

```
LIB "dmod.lib";
ring r = 0,(x,y),Dp;
poly F = x4+y5+x*y4;
printlevel = 0;
def A  = Sannfslog(F);
setring A;
LD1;
↦ LD1[1]=4*x^2*Dx+5*x*y*Dx+3*x*y*Dy+4*y^2*Dy-16*x*s-20*y*s
↦ LD1[2]=16*x*y^2*Dx+4*y^3*Dx+12*y^3*Dy-125*x*y*Dx-4*x^2*Dy+5*x*y*Dy-100*y^\
   2*Dy-64*y^2*s+500*y*s
```

7.7.4.5 bernsteinBM

Procedure from library `dmod.lib` (see Section 7.7.4 [dmod_lib], page 400).

Usage: bernsteinBM(f [,eng]); f a poly, eng an optional int

Return: list (of roots of the Bernstein polynomial b and their multiplicies)

Purpose: compute the global Bernstein-Sato polynomial for a hypersurface,
defined by f, according to the algorithm by Briancon and Maisonobe

Note: If eng <>0, `std` is used for Groebner basis computations,
otherwise, and by default `slimgb` is used.

Display: If `printlevel`=1, progress debug messages will be printed,
if `printlevel`>=2, all the debug messages will be printed.

Example:

```
LIB "dmod.lib";
ring r = 0,(x,y,z,w),Dp;
poly F = x^3+y^3+z^2*w;
printlevel = 0;
```

```
bernsteinBM(F);
↦ [1]:
↦    _[1]=-7/6
↦    _[2]=-5/3
↦    _[3]=-1
↦    _[4]=-2
↦    _[5]=-3/2
↦    _[6]=-11/6
↦    _[7]=-7/3
↦ [2]:
↦    1,1,1,1,1,1,1
```

7.7.4.6 bernsteinLift

Procedure from library `dmod.lib` (see Section 7.7.4 [dmod_lib], page 400).

Usage: bernsteinLift(I, F [,eng]); I an ideal, F a poly, eng an optional int

Return: list

Purpose: compute the (multiple of) Bernstein-Sato polynomial with lift-like method, based on the output of Sannfs-like procedure

Note: the output list contains the roots with multiplicities of the candidate for being Bernstein-Sato polynomial of f.
If eng `<>`0, `std` is used for Groebner basis computations, otherwise and by default `slimgb` is used.
If printlevel=1, progress debug messages will be printed, if printlevel>=2, all the debug messages will be printed.

Example:

```
LIB "dmod.lib";
ring r = 0,(x,y,z),Dp;
poly F = x^3+y^3+z^3;
printlevel = 0;
def A = Sannfs(F);   setring A;
LD;
↦ LD[1]=z^2*Dy-y^2*Dz
↦ LD[2]=x*Dx+y*Dy+z*Dz-3*s
↦ LD[3]=z^2*Dx-x^2*Dz
↦ LD[4]=y^2*Dx-x^2*Dy
poly F = imap(r,F);
list L  = bernsteinLift(LD,F); L;
↦ [1]:
↦    _[1]=-1
↦    _[2]=-2
↦    _[3]=-4/3
↦    _[4]=-5/3
↦ [2]:
↦    2,1,1,1
poly bs = fl2poly(L,"s"); bs; // the candidate for Bernstein-Sato polynomial
↦ s^5+7*s^4+173/9*s^3+233/9*s^2+154/9*s+40/9
```

7.7.4.7 operatorBM

Procedure from library `dmod.lib` (see Section 7.7.4 [dmod_lib], page 400).

Usage: operatorBM(f [,eng]); f a poly, eng an optional int

Return: ring

Purpose: compute the B-operator and other relevant data for Ann F^s,
using e.g. algorithm by Briancon and Maisonobe for Ann F^s and BS.

Note: activate the output ring with the `setring` command. In the output ring D[s]
- the polynomial F is the same as the input,
- the ideal LD is the annihilator of f^s in Dn[s],
- the ideal LD0 is the needed D-mod structure, where LD0 = LD|s=s0,
- the polynomial bs is the global Bernstein polynomial of f in the variable s,
- the list BS contains all the roots with multiplicities of the global Bernstein polynomial of f,
- the polynomial PS is an operator in Dn[s] such that PS*f^(s+1) = bs*f^s.

If eng <>0, `std` is used for Groebner basis computations,
otherwise and by default `slimgb` is used.

Display: If `printlevel`=1, progress debug messages will be printed,
if `printlevel`>=2, all the debug messages will be printed.

Example:

```
LIB "dmod.lib";
ring r = 0,(x,y,z),Dp;
poly F = x^3+y^3+z^3;
printlevel = 0;
def A = operatorBM(F);
setring A;
F; // the original polynomial itself
↦ x^3+y^3+z^3
LD; // generic annihilator
↦ LD[1]=x*Dx+y*Dy+z*Dz-3*s
↦ LD[2]=z^2*Dy-y^2*Dz
↦ LD[3]=z^2*Dx-x^2*Dz
↦ LD[4]=y^2*Dx-x^2*Dy
↦ LD[5]=x^3*Dz+y^3*Dz+z^3*Dz-3*z^2*s
↦ LD[6]=x^3*Dy+y^3*Dy+y^2*z*Dz-3*y^2*s
LD0; // annihilator
↦ LD0[1]=x*Dx+y*Dy+z*Dz+6
↦ LD0[2]=z^2*Dy-y^2*Dz
↦ LD0[3]=z^2*Dx-x^2*Dz
↦ LD0[4]=y^2*Dx-x^2*Dy
↦ LD0[5]=x^3*Dz+y^3*Dz+z^3*Dz+6*z^2
↦ LD0[6]=x^3*Dy+y^3*Dy+y^2*z*Dz+6*y^2
bs; // normalized Bernstein poly
↦ s^5+7*s^4+173/9*s^3+233/9*s^2+154/9*s+40/9
BS; // roots and multiplicities of the Bernstein poly
↦ [1]:
↦    _[1]=-1
↦    _[2]=-4/3
↦    _[3]=-5/3
↦    _[4]=-2
↦ [2]:
↦    2,1,1,1
PS; // the operator, s.t. PS*F^{s+1} = bs*F^s mod LD
```

```
↦ 2/81*y*z*Dx^3*Dy*Dz-2/81*y*z*Dy^4*Dz-4/81*y^2*Dy^2*Dz^3-2/81*y*z*Dy*Dz^4+\
   2/81*y*Dx^3*Dy*s-2/81*y*Dy^4*s+2/81*z*Dx^3*Dz*s+2/27*z*Dy^3*Dz*s+2/27*y*D\
   y*Dz^3*s-2/81*z*Dz^4*s+2/27*y*Dx^3*Dy-2/27*y*Dy^4+2/27*z*Dx^3*Dz+2/27*z*D\
   y^3*Dz-10/81*y*Dy*Dz^3-2/27*z*Dz^4+1/27*Dx^3*s^2+1/9*Dy^3*s^2+1/9*Dz^3*s^\
   2+5/27*Dx^3*s+11/27*Dy^3*s+11/27*Dz^3*s+20/81*Dx^3+8/27*Dy^3+16/81*Dz^3
reduce(PS*F-bs,LD); // check the property of PS
↦ 0
```

7.7.4.8 operatorModulo

Procedure from library `dmod.lib` (see Section 7.7.4 [dmod_lib], page 400).

Usage: operatorModulo(f,I,b); f a poly, I an ideal, b a poly

Return: poly

Purpose: compute the B-operator from the polynomial f,
ideal I = Ann f^s and Bernstein-Sato polynomial b
using modulo i.e. kernel of module homomorphism

Note: The computations take place in the ring, similar to the one
returned by Sannfs procedure.
Note, that operator is not completely reduced wrt Ann f^{s+1}.
If printlevel=1, progress debug messages will be printed,
if printlevel>=2, all the debug messages will be printed.

Example:

```
LIB "dmod.lib";
//  LIB "dmod.lib"; option(prot); option(mem);
ring r = 0,(x,y),Dp;
poly F = x^3+y^3+x*y^3;
def A = Sannfs(F); // here we get LD = ann f^s
setring A;
poly F = imap(r,F);
def B = annfs0(LD,F); // to obtain BS polynomial
list BS = imap(B,BS);   poly bs = fl2poly(BS,"s");
poly PS = operatorModulo(F,LD,bs);
LD = groebner(LD);
PS = NF(PS,subst(LD,s,s+1));; // reduction modulo Ann s^{s+1}
↦ // ** _ is no standard basis
size(PS);
↦ 56
lead(PS);
↦ -2/243*y^3*Dx*Dy^3
reduce(PS*F-bs,LD); // check the defining property of PS
↦ 0
```

7.7.4.9 annfsParamBM

Procedure from library `dmod.lib` (see Section 7.7.4 [dmod_lib], page 400).

Usage: annfsParamBM(f [,eng]); f a poly, eng an optional int

Return: ring

Purpose: compute the generic Ann F^s and exceptional parametric constellations
of a polynomial with parametric coefficients with the BM algorithm

Note: activate the output ring with the `setring` command. In this ring,
- the ideal LD is the D-module structure oa Ann F^s
- the ideal Param contains special parameters as entries
If eng <>0, `std` is used for Groebner basis computations,
otherwise, and by default `slimgb` is used.

Display: If `printlevel`=1, progress debug messages will be printed,
if `printlevel`>=2, all the debug messages will be printed.

Example:

```
LIB "dmod.lib";
ring r = (0,a,b),(x,y),Dp;
poly F = x^2 - (y-a)*(y-b);
printlevel = 0;
def A = annfsParamBM(F);  setring A;
LD;
↦ LD[1]=2*y*Dx+2*x*Dy+(-a-b)*Dx
↦ LD[2]=x^2*Dy-y^2*Dy+(a+b)*y*Dy+2*y*s+(-a*b)*Dy+(-a-b)*s
↦ LD[3]=4*x^2*Dx+4*x*y*Dy+(-2*a-2*b)*x*Dy-8*x*s+(a^2-2*a*b+b^2)*Dx
Param;
↦ Param[1]=(a-b)
setring r;
poly G = x2-(y-a)^2; // try the exceptional value b=a of parameters
def B = annfsParamBM(G); setring B;
LD;
↦ LD[1]=y*Dx+x*Dy+(-a)*Dx
↦ LD[2]=x*Dx+y*Dy+(-a)*Dy-2*s
↦ LD[3]=x^2*Dy-y^2*Dy+(2*a)*y*Dy+2*y*s+(-a^2)*Dy+(-2*a)*s
Param;
↦ Param[1]=0
```

7.7.4.10 annfsBMI

Procedure from library `dmod.lib` (see Section 7.7.4 [dmod_lib], page 400).

Usage: annfsBMI(F [,eng,met,us]); F an ideal, eng, met, us optional ints

Return: ring

Purpose: compute two kinds of Bernstein-Sato ideals, associated to
f = F[1]*..*F[P], with the multivariate algorithm by Briancon and Maisonobe.

Note: activate the output ring with the `setring` command. In this ring,
- the ideal LD is the annihilator of F[1]^s_1*..*F[P]^s_p,
- the list or ideal BS is a Bernstein-Sato ideal of a polynomial f = F[1]*..*F[P].
If eng <>0, `std` is used for Groebner basis computations,
otherwise, and by default `slimgb` is used.
If met <0, the B-Sigma ideal (cf. Castro and Ucha,
'On the computation of Bernstein-Sato ideals', 2005) is computed.
If 0 < met < P, then the ideal B_P (cf. the paper) is computed.
Otherwise, and by default, the ideal B (cf. the paper) is computed.
If us<>0, then syzygies-driven method is used.
If the output ideal happens to be principal, the list of factors
with their multiplicities is returned instead of the ideal.

If printlevel=1, progress debug messages will be printed,
if printlevel>=2, all the debug messages will be printed.

Example:

```
LIB "dmod.lib";
ring r = 0,(x,y),Dp;
ideal F = x,y,x+y;
printlevel = 0;
// *1* let us compute the B ideal
def A = annfsBMI(F);    setring A;
LD; // annihilator
↦ LD[1]=x*Dx+y*Dy-s(1)-s(2)-s(3)
↦ LD[2]=x*y*Dy+y^2*Dy-x*s(2)-y*s(2)-y*s(3)
↦ LD[3]=y^2*Dx*Dy-y^2*Dy^2+y*Dy*s(1)-y*Dx*s(2)+2*y*Dy*s(2)-y*Dx*s(3)+y*Dy*s\
   (3)-s(1)*s(2)-s(2)^2-s(2)*s(3)-s(2)
BS; // Bernstein-Sato ideal
↦ [1]:
↦    _[1]=s(1)+1
↦    _[2]=s(2)+1
↦    _[3]=s(1)+s(2)+s(3)+2
↦    _[4]=s(1)+s(2)+s(3)+4
↦    _[5]=s(1)+s(2)+s(3)+3
↦    _[6]=s(3)+1
↦ [2]:
↦    1,1,1,1,1,1
// *2* now, let us compute B-Sigma ideal
setring r;
def Sigma = annfsBMI(F,0,-1); setring Sigma;
print(matrix(lead(LD))); // compact form of leading
↦ x*Dx,x*y*Dy,y^2*Dx*Dy
//  monomials from the annihilator
BS; // Bernstein-Sato B-Sigma ideal: it is principal,
↦ [1]:
↦    _[1]=s(1)+s(2)+s(3)+2
↦ [2]:
↦    1
// so factors and multiplicities are returned
// *3* and now, let us compute B-i ideal
setring r;
def Bi = annfsBMI(F,0,3); // that is F[3]=x+y is taken
setring Bi;
print(matrix(lead(LD)));   // compact form of leading
↦ x*Dx,x*y*Dy,y^2*Dx*Dy
//  monomials from the annihilator
BS; // the B_3 ideal: it is principal, so factors
↦ [1]:
↦    _[1]=s(1)+s(2)+s(3)+2
↦    _[2]=s(3)+1
↦ [2]:
↦    1,1
// and multiplicities are returned
```

7.7.4.11 checkRoot

Procedure from library `dmod.lib` (see Section 7.7.4 [dmod_lib], page 400).

Usage: checkRoot(f,alpha [,S,eng]); poly f, number alpha, string S, int eng

Return: int

Assume: Basering is a commutative ring, alpha is a positive rational number.

Purpose: check whether a negative rational number -alpha is a root of the global Bernstein-Sato polynomial of f and compute its multiplicity,
with the algorithm given by S and with the Groebner basis engine given by eng.

Note: The annihilator of f^s in D[s] is needed and hence it is computed with the algorithm by Briancon and Maisonobe. The value of a string S can be
'alg1' (default) - for the algorithm 1 of [LM08]
'alg2' - for the algorithm 2 of [LM08]
Depending on the value of S, the output of type int is:
'alg1': 1 only if -alpha is a root of the global Bernstein-Sato polynomial
'alg2': the multiplicity of -alpha as a root of the global Bernstein-Sato polynomial of f. If -alpha is not a root, the output is 0.
If eng <>0, `std` is used for Groebner basis computations,
otherwise (and by default) `slimgb` is used.

Display: If printlevel=1, progress debug messages will be printed,
if printlevel>=2, all the debug messages will be printed.

Example:

```
LIB "dmod.lib";
printlevel=0;
ring r = 0,(x,y),Dp;
poly F = x^4+y^5+x*y^4;
checkRoot(F,11/20);    // -11/20 is a root of bf
↦ 1
poly G = x*y;
checkRoot(G,1,"alg2"); // -1 is a root of bg with multiplicity 2
↦ 2
```

7.7.4.12 SannfsBFCT

Procedure from library `dmod.lib` (see Section 7.7.4 [dmod_lib], page 400).

Usage: SannfsBFCT(f [,a,b,c]); f a poly, a,b,c optional ints

Return: ring

Purpose: compute a Groebner basis either of Ann(f^s)+<f> or of
Ann(f^s)+<f,f_1,...,f_n> in D[s]

Note: Activate the output ring with the `setring` command.
This procedure, unlike SannfsBM, returns the ring D[s] with an anti-elimination ordering for s.
The output ring contains an ideal `LD`, being a Groebner basis
either of Ann(f^s)+<f>, if a=0 (and by default), or of
Ann(f^s)+<f,f_1,...,f_n>, otherwise.
Here, f_i stands for the i-th partial derivative of f.

If b<>0, `std` is used for Groebner basis computations,
otherwise, and by default `slimgb` is used.
If c<>0, `std` is used for Groebner basis computations of
ideals <I+J> when I is already a Groebner basis of <I>.
Otherwise, and by default the engine determined by the switch b is
used. Note that in the case c<>0, the choice for b will be
overwritten only for the types of ideals mentioned above.
This means that if b<>0, specifying c has no effect.

Display: If printlevel=1, progress debug messages will be printed,
if printlevel>=2, all the debug messages will be printed.

Example:

```
LIB "dmod.lib";
ring r = 0,(x,y,z,w),Dp;
poly F = x^3+y^3+z^3*w;
// compute Ann(F^s)+<F> using slimgb only
def A = SannfsBFCT(F);
setring A; A;
↦ // coefficients: QQ
↦ // number of vars : 9
↦ //        block   1 : ordering dp
↦ //                  : names    s
↦ //        block   2 : ordering dp
↦ //                  : names    x y z w Dx Dy Dz Dw
↦ //        block   3 : ordering C
↦ // noncommutative relations:
↦ //    Dxx=x*Dx+1
↦ //    Dyy=y*Dy+1
↦ //    Dzz=z*Dz+1
↦ //    Dww=w*Dw+1
LD;
↦ LD[1]=z*Dz-3*w*Dw
↦ LD[2]=3*s-x*Dx-y*Dy-3*w*Dw
↦ LD[3]=y^2*Dx-x^2*Dy
↦ LD[4]=z^2*w*Dy-y^2*Dz
↦ LD[5]=z^3*Dy-3*y^2*Dw
↦ LD[6]=z^2*w*Dx-x^2*Dz
↦ LD[7]=z^3*Dx-3*x^2*Dw
↦ LD[8]=z^3*w+x^3+y^3
↦ LD[9]=x^3*Dy+y^3*Dy+3*y^2*w*Dw+3*y^2
↦ LD[10]=x^3*Dx+x^2*y*Dy+3*x^2*w*Dw+3*x^2
↦ LD[11]=3*z*w^2*Dy*Dw-y^2*Dz^2+2*z*w*Dy
↦ LD[12]=3*z*w^2*Dx*Dw-x^2*Dz^2+2*z*w*Dx
↦ LD[13]=3*z^2*w^2*Dw+x^3*Dz+y^3*Dz+3*z^2*w
↦ LD[14]=9*w^3*Dy*Dw^2-y^2*Dz^3+18*w^2*Dy*Dw+2*w*Dy
↦ LD[15]=9*w^3*Dx*Dw^2-x^2*Dz^3+18*w^2*Dx*Dw+2*w*Dx
↦ LD[16]=9*z*w^3*Dw^2+x^3*Dz^2+y^3*Dz^2+24*z*w^2*Dw+6*z*w
↦ LD[17]=27*w^4*Dw^3+x^3*Dz^3+y^3*Dz^3+135*w^3*Dw^2+114*w^2*Dw+6*w
// the Bernstein-Sato poly of F:
vec2poly(pIntersect(s,LD));
↦ s^6+28/3*s^5+320/9*s^4+1910/27*s^3+2093/27*s^2+1198/27*s+280/27
// a fancier example:
```

```
def R = reiffen(4,5); setring R;
RC; // the Reiffen curve in 4,5
↦ xy4+y5+x4
// compute Ann(RC^s)+<RC,diff(RC,x),diff(RC,y)>
// using std for GB computations of ideals <I+J>
// where I is already a GB of <I>
// and slimgb for other ideals
def B = SannfsBFCT(RC,1,0,1);
setring B;
// the Bernstein-Sato poly of RC:
(s-1)*vec2poly(pIntersect(s,LD));
↦ s^13+10*s^12+44*s^11+44099/400*s^10+13355001/80000*s^9+22138611/160000*s^\
   8+1747493/160000*s^7-7874303503/64000000*s^6-4244944536107/25600000000*s^\
   5-3066298289417/25600000000*s^4-2787777479229/51200000000*s^3-19980507461\
   787/1280000000000*s^2-663659243177931/256000000000000*s-48839201079669/25\
   6000000000000
```

7.7.4.13 annfs0

Procedure from library `dmod.lib` (see Section 7.7.4 [dmod_lib], page 400).

Usage: annfs0(I, F [,eng]); I an ideal, F a poly, eng an optional int

Return: ring

Purpose: compute the annihilator ideal of f^s in the Weyl Algebra, based on the output of Sannfs-like procedure

Note: activate the output ring with the `setring` command. In this ring,
- the ideal LD (which is a Groebner basis) is the annihilator of f^s,
- the list BS contains the roots with multiplicities of BS polynomial of f.
If eng <>0, `std` is used for Groebner basis computations,
otherwise and by default `slimgb` is used.
If printlevel=1, progress debug messages will be printed,
if printlevel>=2, all the debug messages will be printed.

Example:

```
LIB "dmod.lib";
ring r = 0,(x,y,z),Dp;
poly F = x^3+y^3+z^3;
printlevel = 0;
def A = SannfsBM(F);   setring A;
// alternatively, one can use SannfsOT or SannfsLOT
LD;
↦ LD[1]=z^2*Dy-y^2*Dz
↦ LD[2]=x*Dx+y*Dy+z*Dz-3*s
↦ LD[3]=z^2*Dx-x^2*Dz
↦ LD[4]=y^2*Dx-x^2*Dy
poly F = imap(r,F);
def B  = annfs0(LD,F);  setring B;
LD;
↦ LD[1]=x*Dx+y*Dy+z*Dz+6
↦ LD[2]=z^2*Dy-y^2*Dz
↦ LD[3]=z^2*Dx-x^2*Dz
↦ LD[4]=y^2*Dx-x^2*Dy
```

```
↦ LD[5]=x^3*Dz+y^3*Dz+z^3*Dz+6*z^2
↦ LD[6]=x^3*Dy+y^3*Dy+y^2*z*Dz+6*y^2
BS;
↦ [1]:
↦    _[1]=-1
↦    _[2]=-2
↦    _[3]=-5/3
↦    _[4]=-4/3
↦ [2]:
↦    2,1,1,1
```

7.7.4.14 annfs2

Procedure from library `dmod.lib` (see Section 7.7.4 [dmod_lib], page 400).

Usage: annfs2(I, F [,eng]); I an ideal, F a poly, eng an optional int

Return: ring

Purpose: compute the annihilator ideal of f^s in the Weyl Algebra,
based on the output of Sannfs-like procedure
annfs2 uses shorter expressions in the variable s (the idea of Noro).

Note: activate the output ring with the `setring` command. In this ring,
- the ideal LD (which is a Groebner basis) is the annihilator of f^s,
- the list BS contains the roots with multiplicities of the BS polynomial.
If eng <>0, `std` is used for Groebner basis computations,
otherwise and by default `slimgb` is used.

Display: If `printlevel`=1, progress debug messages will be printed,
if `printlevel`>=2, all the debug messages will be printed.

Example:

```
LIB "dmod.lib";
ring r = 0,(x,y,z),Dp;
poly F = x^3+y^3+z^3;
printlevel = 0;
def A = SannfsBM(F);
setring A;
LD;
↦ LD[1]=z^2*Dy-y^2*Dz
↦ LD[2]=x*Dx+y*Dy+z*Dz-3*s
↦ LD[3]=z^2*Dx-x^2*Dz
↦ LD[4]=y^2*Dx-x^2*Dy
poly F = imap(r,F);
def B  = annfs2(LD,F);
setring B;
LD;
↦ LD[1]=x*Dx+y*Dy+z*Dz+6
↦ LD[2]=z^2*Dy-y^2*Dz
↦ LD[3]=z^2*Dx-x^2*Dz
↦ LD[4]=y^2*Dx-x^2*Dy
↦ LD[5]=x^3*Dz+y^3*Dz+z^3*Dz+6*z^2
↦ LD[6]=x^3*Dy+y^3*Dy+y^2*z*Dz+6*y^2
BS;
```

```
↦ [1]:
↦    _[1]=-1
↦    _[2]=-2
↦    _[3]=-5/3
↦    _[4]=-4/3
↦ [2]:
↦    2,1,1,1
```

7.7.4.15 annfsRB

Procedure from library `dmod.lib` (see Section 7.7.4 [dmod_lib], page 400).

Usage: annfsRB(I, F [,eng]); I an ideal, F a poly, eng an optional int

Return: ring

Purpose: compute the annihilator ideal of f^s in the Weyl Algebra,
based on the output of Sannfs like procedure

Note: activate the output ring with the `setring` command. In this ring,
- the ideal LD (which is a Groebner basis) is the annihilator of f^s,
- the list BS contains the roots with multiplicities of a Bernstein polynomial of f.
If eng <>0, `std` is used for Groebner basis computations,
otherwise and by default `slimgb` is used.
This procedure uses in addition to F its Jacobian ideal.

Display: If `printlevel`=1, progress debug messages will be printed,
if `printlevel`>=2, all the debug messages will be printed.

Example:

```
LIB "dmod.lib";
ring r = 0,(x,y,z),Dp;
poly F = x^3+y^3+z^3;
printlevel = 0;
def A = SannfsBM(F); setring A;
LD; // s-parametric ahhinilator
↦ LD[1]=z^2*Dy-y^2*Dz
↦ LD[2]=x*Dx+y*Dy+z*Dz-3*s
↦ LD[3]=z^2*Dx-x^2*Dz
↦ LD[4]=y^2*Dx-x^2*Dy
poly F = imap(r,F);
def B  = annfsRB(LD,F); setring B;
LD;
↦ LD[1]=x*Dx+y*Dy+z*Dz+6
↦ LD[2]=z^2*Dy-y^2*Dz
↦ LD[3]=z^2*Dx-x^2*Dz
↦ LD[4]=y^2*Dx-x^2*Dy
↦ LD[5]=x^3*Dz+y^3*Dz+z^3*Dz+6*z^2
↦ LD[6]=x^3*Dy+y^3*Dy+y^2*z*Dz+6*y^2
BS;
↦ [1]:
↦    _[1]=-1
↦    _[2]=-2
↦    _[3]=-5/3
↦    _[4]=-4/3
```

```
↦ [2]:
↦    2,1,1,1
```

7.7.4.16 checkFactor

Procedure from library `dmod.lib` (see Section 7.7.4 [dmod_lib], page 400).

Usage: checkFactor(I,f,qs [,eng]); I an ideal, f a poly, qs a poly, eng an optional int

Assume: checkFactor is called from the basering, created by Sannfs-like proc,
that is, from the Weyl algebra in x1,..,xN,d1,..,dN tensored with K[s].
The ideal I is the annihilator of f^s in D[s], that is the ideal, computed
by Sannfs-like procedure (usually called LD there).
Moreover, f is a polynomial in K[x1,..,xN] and qs is a polynomial in K[s].

Return: int, 1 if qs is a factor of the global Bernstein polynomial of f and 0 otherwise

Purpose: check whether a univariate polynomial qs is a factor of the
Bernstein-Sato polynomial of f without explicit knowledge of the latter.

Note: If eng <>0, `std` is used for Groebner basis computations,
otherwise (and by default) `slimgb` is used.

Display: If printlevel=1, progress debug messages will be printed,
if printlevel>=2, all the debug messages will be printed.

Example:

```
LIB "dmod.lib";
ring r = 0,(x,y),Dp;
poly F = x^4+y^5+x*y^4;
printlevel = 0;
def A = Sannfs(F);
setring A;
poly F = imap(r,F);
checkFactor(LD,F,20*s+31);     // -31/20 is not a root of bs
↦ 0
checkFactor(LD,F,20*s+11);     // -11/20 is a root of bs
↦ 1
checkFactor(LD,F,(20*s+11)^2); // the multiplicity of -11/20 is 1
↦ 0
```

7.7.4.17 arrange

Procedure from library `dmod.lib` (see Section 7.7.4 [dmod_lib], page 400).

Usage: arrange(p); int p

Return: ring

Purpose: set up the polynomial, describing a hyperplane arrangement

Note: must be executed in a commutative ring

Assume: basering is present and it is commutative

Example:

```
LIB "dmod.lib";
ring X = 0,(x,y,z,t),dp;
poly q = arrange(3);
factorize(q,1);
↦ _[1]=x
↦ _[2]=x+z
↦ _[3]=y
↦ _[4]=y+z
↦ _[5]=x+y
↦ _[6]=x+y+z
↦ _[7]=z
```

7.7.4.18 reiffen

Procedure from library `dmod.lib` (see Section 7.7.4 [dmod_lib], page 400).

Usage: reiffen(p, q); int p, int q

Return: ring

Purpose: set up the polynomial, describing a Reiffen curve

Note: activate the output ring with the `setring` command and
find the curve as a polynomial RC.
A Reiffen curve is defined as RC = x^p + y^q + xy^{q-1}, q >= p+1 >= 5

Example:

```
LIB "dmod.lib";
def r = reiffen(4,5);
setring r;
RC;
↦ xy4+y5+x4
```

7.7.4.19 isHolonomic

Procedure from library `dmod.lib` (see Section 7.7.4 [dmod_lib], page 400).

Usage: isHolonomic(M); M an ideal/module/matrix

Return: int, 1 if M is holonomic over the base ring, and 0 otherwise

Assume: basering is a Weyl algebra in characteristic 0

Purpose: check whether M is holonomic over the base ring

Note: M is holonomic if 2*dim(M) = dim(R), where R is the
base ring; dim stands for Gelfand-Kirillov dimension

Example:

```
LIB "dmod.lib";
ring R = 0,(x,y),dp;
poly F = x*y*(x+y);
def A = annfsBM(F,0);
setring A;
LD;
↦ LD[1]=x*Dx+y*Dy+3
↦ LD[2]=x*y*Dy+y^2*Dy+x+2*y
↦ LD[3]=y^2*Dx*Dy-y^2*Dy^2+2*y*Dx-4*y*Dy-2
```

```
isHolonomic(LD);
↦ 1
ideal I = std(LD[1]);
I;
↦ I[1]=x*Dx+y*Dy+3
isHolonomic(I);
↦ 0
```

7.7.4.20 convloc

Procedure from library `dmod.lib` (see Section 7.7.4 [dmod_lib], page 400).

Usage: convloc(L); L a list

Return: list

Purpose: convert a ringlist L into another ringlist,
where all the 'p' orderings are replaced with the 's' orderings, e.g. `dp` by `ds`.

Assume: L is a result of a ringlist command

Example:

```
LIB "dmod.lib";
ring r = 0,(x,y,z),(Dp(2),dp(1));
list L = ringlist(r);
list N = convloc(L);
def rs = ring(N);
setring rs;
rs;
↦ // coefficients: QQ
↦ // number of vars : 3
↦ //        block   1 : ordering Ds
↦ //                  : names    x y
↦ //        block   2 : ordering ls
↦ //                  : names    z
↦ //        block   3 : ordering C
```

7.7.4.21 minIntRoot

Procedure from library `dmod.lib` (see Section 7.7.4 [dmod_lib], page 400).

Usage: minIntRoot(P, fact); P an ideal, fact an int

Return: int

Purpose: minimal integer root of a maximal ideal P

Note: if fact==1, P is the result of some 'factorize' call,
else P is treated as the result of bernstein::gmssing.lib
in both cases without constants and multiplicities

Example:

```
LIB "dmod.lib";
ring r   = 0,(x,y),ds;
poly f1  = x*y*(x+y);
ideal I1 = bernstein(f1)[1]; // a local Bernstein poly
I1;
```

```
↦ I1[1]=-4/3
↦ I1[2]=-1
↦ I1[3]=-2/3
minIntRoot(I1,0);
↦ -1
poly  f2  = x2-y3;
ideal I2  = bernstein(f2)[1];
I2;
↦ I2[1]=-7/6
↦ I2[2]=-1
↦ I2[3]=-5/6
minIntRoot(I2,0);
↦ -1
// now we illustrate the behaviour of factorize
// together with a global ordering
ring r2  = 0,x,dp;
poly f3   = 9*(x+2/3)*(x+1)*(x+4/3); //global b-polynomial of f1=x*y*(x+y)
ideal I3 = factorize(f3,1);
I3;
↦ I3[1]=3x+4
↦ I3[2]=3x+2
↦ I3[3]=x+1
minIntRoot(I3,1);
↦ -1
// and a more interesting situation
ring  s  = 0,(x,y,z),ds;
poly  f  = x3 + y3 + z3;
ideal I  = bernstein(f)[1];
I;
↦ I[1]=-2
↦ I[2]=-5/3
↦ I[3]=-4/3
↦ I[4]=-1
minIntRoot(I,0);
↦ -2
```

7.7.4.22 isRational

Procedure from library `dmod.lib` (see Section 7.7.4 [dmod_lib], page 400).

Usage: isRational(n); n number

Return: int

Purpose: determine whether n is a rational number,
that is it does not contain parameters.

Assume: ground field is of characteristic 0

Example:

```
LIB "dmod.lib";
ring r = (0,a),(x,y),dp;
number n1 = 11/73;
isRational(n1);
↦ 1
```

```
number n2 = (11*a+3)/72;
isRational(n2);
↦ 0
```

7.7.5 dmodapp_lib

Library: dmodapp.lib

Purpose: Applications of algebraic D-modules

Authors: Viktor Levandovskyy, levandov@math.rwth-aachen.de
Daniel Andres, daniel.andres@math.rwth-aachen.de

Support: DFG Graduiertenkolleg 1632 'Experimentelle und konstruktive Algebra'

Overview: Let K be a field of characteristic 0, R = K[x1,...,xN] and D be the Weyl algebra in variables x1,...,xN,d1,...,dN. In this library there are the following procedures for algebraic D-modules:

- given a cyclic representation D/I of a holonomic module and a polynomial F in R, it is proved that the localization of D/I with respect to the mult. closed set of all powers of F is a holonomic D-module. Thus we aim to compute its cyclic representaion D/L for an ideal L in D. The procedures for the localization are DLoc, SDLoc and DLoc0.

- annihilator in D of a given polynomial F from R as well as of a given rational function G/F from Quot(R). These can be computed via procedures annPoly resp. annRat.

- Groebner bases with respect to weights (according to (SST), given an arbitrary integer vector containing weights for variables, one computes the homogenization of a given ideal relative to this vector, then one computes a Groebner basis and returns the dehomogenization of the result), initial forms and initial ideals in Weyl algebras with respect to a given weight vector can be computed with GBWeight, inForm, initialMalgrange and initialIdealW.

- restriction and integration of a holonomic module D/I. Suppose I annihilates a function F(x1,...,xn). Our aim is to compute an ideal J directly from I, which annihilates
- F(0,...,0,xk,...,xn) in case of restriction or
- the integral of F with respect to x1,...,xm in case of integration. The corresponding procedures are restrictionModule, restrictionIdeal, integralModule and integralIdeal.

- characteristic varieties defined by ideals in Weyl algebras can be computed with charVariety and charInfo.

- appelF1, appelF2 and appelF4 return ideals in parametric Weyl algebras, which annihilate corresponding Appel hypergeometric functions.

References:

(SST) Saito, Sturmfels, Takayama 'Groebner Deformations of Hypergeometric Differential Equations', Springer, 2000
(OTW) Oaku, Takayama, Walther 'A Localization Algorithm for D-modules', Journal of Symbolic Computation, 2000

(OT) Oaku, Takayama 'Algorithms for D-modules', Journal of Pure and Applied Algebra, 1998

Procedures: D-module See also: Section 7.7.2 [bfun_lib], page 375; Section 7.7.4 [dmod_lib], page 400; Section 7.7.6 [dmodvar_lib], page 445; Section D.6.9 [gmssing_lib], page 778.

7.7.5.1 annPoly

Procedure from library `dmodapp.lib` (see Section 7.7.5 [dmodapp_lib], page 419).

Usage: annPoly(f); f a poly

Return: ring (a Weyl algebra) containing an ideal 'LD'

Purpose: compute the complete annihilator ideal of f in the corresponding Weyl algebra

Assume: basering is commutative and over a field of characteristic 0

Note: Activate the output ring with the `setring` command.
In the output ring, the ideal 'LD' (in Groebner basis) is the annihilator.

Display: If printlevel =1, progress debug messages will be printed,
if printlevel>=2, all the debug messages will be printed.

Example:

```
LIB "dmodapp.lib";
ring r = 0,(x,y,z),dp;
poly f = x^2*z - y^3;
def A = annPoly(f);
setring A;    // A is the 3rd Weyl algebra in 6 variables
LD;           // the Groebner basis of annihilator
↦ LD[1]=Dz^2
↦ LD[2]=Dy*Dz
↦ LD[3]=Dx*Dy
↦ LD[4]=y*Dy+3*z*Dz-3
↦ LD[5]=x*Dx-2*z*Dz
↦ LD[6]=z*Dx*Dz-Dx
↦ LD[7]=Dy^3+3*Dx^2*Dz
↦ LD[8]=x*Dy^2+3*y*Dx*Dz
↦ LD[9]=x^2*Dy+3*y^2*Dz
↦ LD[10]=Dx^3
↦ LD[11]=3*y*Dx^2+z*Dy^2
↦ LD[12]=3*y^2*Dx+2*x*z*Dy
↦ LD[13]=y^3*Dz-x^2*z*Dz+x^2
gkdim(LD);    // must be 3 = 6/2, since A/LD is holonomic module
↦ 3
NF(Dy^4, LD); // must be 0 since Dy^4 clearly annihilates f
↦ 0
poly f = imap(r,f);
NF(LD*f,std(ideal(Dx,Dy,Dz))); // must be zero if LD indeed annihilates f
↦ _[1]=0
↦ _[2]=0
↦ _[3]=0
↦ _[4]=0
```

```
↦ _[5]=0
↦ _[6]=0
↦ _[7]=0
↦ _[8]=0
↦ _[9]=0
↦ _[10]=0
↦ _[11]=0
↦ _[12]=0
↦ _[13]=0
```

See also: Section 7.7.5.2 [annRat], page 421.

7.7.5.2 annRat

Procedure from library `dmodapp.lib` (see Section 7.7.5 [dmodapp_lib], page 419).

Usage: annRat(g,f); f, g polynomials

Return: ring (a Weyl algebra) containing an ideal 'LD'

Purpose: compute the annihilator of the rational function g/f in the corresponding Weyl algebra

Assume: basering is commutative and over a field of characteristic 0

Note: Activate the output ring with the `setring` command.
In the output ring, the ideal 'LD' (in Groebner basis) is the annihilator of g/f.
The algorithm uses the computation of Ann(f^{-1}) via D-modules, see (SST).

Display: If printlevel =1, progress debug messages will be printed, if printlevel>=2, all the debug messages will be printed.

Example:

```
LIB "dmodapp.lib";
ring r = 0,(x,y),dp;
poly g = 2*x*y;  poly f = x^2 - y^3;
def B = annRat(g,f);
setring B;
LD;
↦ LD[1]=3*y^2*Dx^2*Dy+2*x*Dx*Dy^2+9*y*Dx^2+4*Dy^2
↦ LD[2]=3*y^3*Dx^2-10*x*y*Dx*Dy-8*y^2*Dy^2+10*x*Dx
↦ LD[3]=y^3*Dy^2-x^2*Dy^2-6*x*y*Dx+2*y^2*Dy+4*y
↦ LD[4]=3*x*Dx+2*y*Dy+1
↦ LD[5]=y^4*Dy-x^2*y*Dy+2*y^3+x^2
// Now, compare with the output of Macaulay2:
ideal tst = 3*x*Dx + 2*y*Dy + 1, y^3*Dy^2 - x^2*Dy^2 + 6*y^2*Dy + 6*y,
9*y^2*Dx^2*Dy-4*y*Dy^3+27*y*Dx^2+2*Dy^2, 9*y^3*Dx^2-4*y^2*Dy^2+10*y*Dy -10;
option(redSB); option(redTail);
LD = groebner(LD);
tst = groebner(tst);
print(matrix(NF(LD,tst)));  print(matrix(NF(tst,LD)));
↦ 0,0,0,0,0
↦ 0,0,0,0,0
// So, these two answers are the same
```

See also: Section 7.7.5.1 [annPoly], page 420.

7.7.5.3 DLoc

Procedure from library `dmodapp.lib` (see Section 7.7.5 [dmodapp_lib], page 419).

Usage: DLoc(I, f); I an ideal, f a poly

Return: list of ideal and list

Assume: the basering is a Weyl algebra

Purpose: compute the presentation of the localization of D/I w.r.t. f^s

Note: In the output list L,
- L[1] is an ideal (given as Groebner basis), the presentation of the localization,
- L[2] is a list containing roots with multiplicities of Bernstein polynomial of (D/I)_f.

Display: If printlevel =1, progress debug messages will be printed,
if printlevel>=2, all the debug messages will be printed.

Example:

```
LIB "dmodapp.lib";
ring r = 0,(x,y,Dx,Dy),dp;
def R = Weyl();    setring R; // Weyl algebra in variables x,y,Dx,Dy
poly F = x2-y3;
ideal I = (y^3 - x^2)*Dx - 2*x, (y^3 - x^2)*Dy + 3*y^2; // I = Dx*F, Dy*F;
// I is not holonomic, since its dimension is not 4/2=2
gkdim(I);
↦ 3
list L = DLoc(I, x2-y3);
L[1]; // localized module (R/I)_f is isomorphic to R/LD0
↦ _[1]=3*x*Dx+2*y*Dy+12
↦ _[2]=3*y^2*Dx+2*x*Dy
↦ _[3]=y^3*Dy-x^2*Dy+6*y^2
L[2]; // description of b-function for localization
↦ [1]:
↦    _[1]=0
↦    _[2]=-1/6
↦    _[3]=1/6
↦ [2]:
↦    1,1,1
```

7.7.5.4 SDLoc

Procedure from library `dmodapp.lib` (see Section 7.7.5 [dmodapp_lib], page 419).

Usage: SDLoc(I, f); I an ideal, f a poly

Return: ring (basering extended by a new variable) containing an ideal 'LD'

Purpose: compute a generic presentation of the localization of D/I w.r.t. f^s

Assume: the basering D is a Weyl algebra over a field of characteristic 0

Note: Activate this ring with the `setring` command. In this ring,
the ideal LD (given as Groebner basis) is the presentation of the localization.

Display: If printlevel =1, progress debug messages will be printed,
if printlevel>=2, all the debug messages will be printed.

Example:

```
LIB "dmodapp.lib";
ring r = 0,(x,y,Dx,Dy),dp;
def R = Weyl(); // Weyl algebra on the variables x,y,Dx,Dy
setring R;
poly F = x2-y3;
ideal I = Dx*F, Dy*F;
// note, that I is not holonomic, since it's dimension is not 2
gkdim(I);  // 3, while dim R = 4
↦ 3
def W = SDLoc(I,F);
setring W; // = R[s], where s is a new variable
LD;        // Groebner basis of s-parametric presentation
↦ LD[1]=3*x*Dx*s+2*y*Dy*s-6*s^2+6*s
↦ LD[2]=3*y^2*Dx*s+2*x*Dy*s
↦ LD[3]=y^3*Dy-x^2*Dy-3*y^2*s+3*y^2
↦ LD[4]=y^3*Dx-x^2*Dx+2*x*s-2*x
```

7.7.5.5 DLoc0

Procedure from library `dmodapp.lib` (see Section 7.7.5 [dmodapp_lib], page 419).

Usage: DLoc0(I, f); I an ideal, f a poly

Return: ring (a Weyl algebra) containing an ideal 'LD0' and a list 'BS'

Purpose: compute the presentation of the localization of D/I w.r.t. f^s,
where D is a Weyl Algebra, based on the output of procedure SDLoc

Assume: the basering is similar to the output ring of SDLoc procedure

Note: activate the output ring with the `setring` command. In this ring,
- the ideal LD0 (given as Groebner basis) is the presentation of the localization,
- the list BS contains roots and multiplicities of Bernstein polynomial of (D/I)_f.

Display: If printlevel =1, progress debug messages will be printed,
if printlevel>=2, all the debug messages will be printed.

Example:

```
LIB "dmodapp.lib";
ring r = 0,(x,y,Dx,Dy),dp;
def R = Weyl();    setring R; // Weyl algebra in variables x,y,Dx,Dy
poly F = x2-y3;
ideal I = (y^3 - x^2)*Dx - 2*x, (y^3 - x^2)*Dy + 3*y^2; // I = Dx*F, Dy*F;
// moreover I is not holonomic, since its dimension is not 2 = 4/2
gkdim(I); // 3
↦ 3
def W = SDLoc(I,F);  setring W; // creates ideal LD in W = R[s]
def U = DLoc0(LD, x2-y3);  setring U; // compute in R
LD0; // Groebner basis of the presentation of localization
↦ LD0[1]=3*x*Dx+2*y*Dy+12
```

```
↦ LD0[2]=3*y^2*Dx+2*x*Dy
↦ LD0[3]=y^3*Dy-x^2*Dy+6*y^2
BS; // description of b-function for localization
↦ [1]:
↦    _[1]=0
↦    _[2]=-1/6
↦    _[3]=1/6
↦ [2]:
↦    1,1,1
```

7.7.5.6 GBWeight

Procedure from library `dmodapp.lib` (see Section 7.7.5 [dmodapp_lib], page 419).

Usage: GBWeight(I,u,v [,s,t,w]);
I ideal, u,v intvecs, s,t optional ints, w an optional intvec

Return: ideal, Groebner basis of I w.r.t. the weights u and v

Assume: The basering is the n-th Weyl algebra over a field of characteristic 0 and for all 1<=i<=n the identity var(i+n)*var(i)=var(i)*var(i+1)+1 holds, i.e. the sequence of variables is given by x(1),...,x(n),D(1),...,D(n), where D(i) is the differential operator belonging to x(i).

Purpose: computes a Groebner basis with respect to given weights

Note: The weights u and v are understood as weight vectors for x(i) and D(i), respectively. According to (SST), one computes the homogenization of a given ideal relative to (u,v), then one computes a Groebner basis and returns the dehomogenization of the result.
If s<>0, `std` is used for Groebner basis computations, otherwise, and by default, `slimgb` is used.
If t<>0, a matrix ordering is used for Groebner basis computations, otherwise, and by default, a block ordering is used.
If w is given and consists of exactly 2*n strictly positive entries, w is used for constructing the weighted homogenized Weyl algebra, see Noro (2002). Otherwise, and by default, the homogenization weight (1,...,1) is used.

Display: If printlevel=1, progress debug messages will be printed, if printlevel>=2, all the debug messages will be printed.

Example:

```
LIB "dmodapp.lib";
ring r = 0,(x,y,Dx,Dy),dp;
def D2 = Weyl();
setring D2;
ideal I = 3*x^2*Dy+2*y*Dx,2*x*Dx+3*y*Dy+6;
intvec u = -2,-3;
intvec v = -u;
GBWeight(I,u,v);
↦ _[1]=2*x*Dx+3*y*Dy+6
↦ _[2]=3*x^2*Dy+2*y*Dx
↦ _[3]=9*x*y*Dy^2-4*y*Dx^2+15*x*Dy
```

```
↦ _[4]=27*y^2*Dy^3+8*y*Dx^3+135*y*Dy^2+105*Dy
ideal J = std(I);
GBWeight(J,u,v); // same as above
↦ _[1]=2*x*Dx+3*y*Dy+6
↦ _[2]=3*x^2*Dy+2*y*Dx
↦ _[3]=9*x*y*Dy^2-4*y*Dx^2+15*x*Dy
↦ _[4]=27*y^2*Dy^3+8*y*Dx^3+135*y*Dy^2+105*Dy
u = 0,1;
GBWeight(I,u,v);
↦ _[1]=2*x*Dx+3*y*Dy+6
↦ _[2]=3*x^2*Dy+2*y*Dx
↦ _[3]=-x^3*Dx+y^2*Dx-3*x^2
```

7.7.5.7 initialMalgrange

Procedure from library `dmodapp.lib` (see Section 7.7.5 [dmodapp_lib], page 419).

Usage: initialMalgrange(f,[,a,b,v]); f poly, a,b optional ints, v opt. intvec

Return: ring, Weyl algebra induced by basering, extended by two new vars t,Dt

Purpose: computes the initial Malgrange ideal of a given polynomial w.r.t. the weight vector (-1,0...,0,1,0,...,0) such that the weight of t is -1 and the weight of Dt is 1.

Assume: The basering is commutative and over a field of characteristic 0.

Note: Activate the output ring with the `setring` command.
The returned ring contains the ideal 'inF', being the initial ideal of the Malgrange ideal of f.
Varnames of the basering should not include t and Dt.
If a<>0, `std` is used for Groebner basis computations, otherwise, and by default, `slimgb` is used.
If b<>0, a matrix ordering is used for Groebner basis computations, otherwise, and by default, a block ordering is used.
If a positive weight vector v is given, the weight (d,v[1],...,v[n],1,d+1-v[1],...,d+1-v[n]) is used for homogenization computations, where d denotes the weighted degree of f.
Otherwise and by default, v is set to (1,...,1). See Noro (2002).

Display: If printlevel=1, progress debug messages will be printed, if printlevel>=2, all the debug messages will be printed.

Example:

```
LIB "dmodapp.lib";
ring r = 0,(x,y),dp;
poly f = x^2+y^3+x*y^2;
def D = initialMalgrange(f);
setring D;
inF;
↦ inF[1]=x*Dt
↦ inF[2]=2*x*y*Dx+3*y^2*Dx-y^2*Dy-2*x*Dy
↦ inF[3]=2*x^2*Dx+x*y*Dx+x*y*Dy+18*t*Dt+9*x*Dx-x*Dy+6*y*Dy+4*x+18
↦ inF[4]=18*t*Dt^2+6*y*Dt*Dy-y*Dt+27*Dt
↦ inF[5]=y^2*Dt
↦ inF[6]=2*t*y*Dt+2*x*y*Dx+2*y^2*Dx-6*t*Dt-3*x*Dx-x*Dy-2*y*Dy+2*y-6
```

```
↦ inF[7]=x*y^2+y^3+x^2
↦ inF[8]=2*y^3*Dx-2*y^3*Dy-3*y^2*Dx-2*x*y*Dy+y^2*Dy-4*y^2+36*t*Dt+18*x*Dx+1\
   2*y*Dy+36
setring r;
intvec v = 3,2;
def D2 = initialMalgrange(f,1,1,v);
setring D2;
inF;
↦ inF[1]=x*Dt
↦ inF[2]=2*x*y*Dx+3*y^2*Dx-y^2*Dy-2*x*Dy
↦ inF[3]=4*x^2*Dx-3*y^2*Dx+2*x*y*Dy+y^2*Dy+36*t*Dt+18*x*Dx+12*y*Dy+8*x+36
↦ inF[4]=18*t*Dt^2+6*y*Dt*Dy-y*Dt+27*Dt
↦ inF[5]=y^2*Dt
↦ inF[6]=2*t*y*Dt-y^2*Dx+y^2*Dy-6*t*Dt-3*x*Dx+x*Dy-2*y*Dy+2*y-6
↦ inF[7]=x*y^2+y^3+x^2
↦ inF[8]=2*y^3*Dx-2*y^3*Dy-3*y^2*Dx-2*x*y*Dy+y^2*Dy-4*y^2+36*t*Dt+18*x*Dx+1\
   2*y*Dy+36
```

7.7.5.8 initialIdealW

Procedure from library `dmodapp.lib` (see Section 7.7.5 [dmodapp_lib], page 419).

Usage: initialIdealW(I,u,v [,s,t,w]);
I ideal, u,v intvecs, s,t optional ints, w an optional intvec

Return: ideal, GB of initial ideal of the input ideal wrt the weights u and v

Assume: The basering is the n-th Weyl algebra in characteristic 0 and for all
1<=i<=n the identity var(i+n)*var(i)=var(i)*var(i+1)+1 holds, i.e. the
sequence of variables is given by x(1),...,x(n),D(1),...,D(n),
where D(i) is the differential operator belonging to x(i).

Purpose: computes the initial ideal with respect to given weights.

Note: u and v are understood as weight vectors for x(1..n) and D(1..n)
respectively.
If s<>0, `std` is used for Groebner basis computations,
otherwise, and by default, `slimgb` is used.
If t<>0, a matrix ordering is used for Groebner basis computations,
otherwise, and by default, a block ordering is used.
If w is given and consists of exactly 2*n strictly positive entries,
w is used as homogenization weight.
Otherwise, and by default, the homogenization weight (1,...,1) is used.

Display: If printlevel=1, progress debug messages will be printed,
if printlevel>=2, all the debug messages will be printed.

Example:

```
LIB "dmodapp.lib";
ring r = 0,(x,y,Dx,Dy),dp;
def D2 = Weyl();
setring D2;
ideal I = 3*x^2*Dy+2*y*Dx,2*x*Dx+3*y*Dy+6;
intvec u = -2,-3;
intvec v = -u;
initialIdealW(I,u,v);
```

```
↦ _[1]=2*x*Dx+3*y*Dy+6
↦ _[2]=3*x^2*Dy+2*y*Dx
↦ _[3]=9*x*y*Dy^2-4*y*Dx^2+15*x*Dy
↦ _[4]=27*y^2*Dy^3+8*y*Dx^3+135*y*Dy^2+105*Dy
ideal J = std(I);
initialIdealW(J,u,v); // same as above
↦ _[1]=2*x*Dx+3*y*Dy+6
↦ _[2]=3*x^2*Dy+2*y*Dx
↦ _[3]=9*x*y*Dy^2-4*y*Dx^2+15*x*Dy
↦ _[4]=27*y^2*Dy^3+8*y*Dx^3+135*y*Dy^2+105*Dy
u = 0,1;
initialIdealW(I,u,v);
↦ _[1]=Dx
↦ _[2]=Dy
```

7.7.5.9 inForm

Procedure from library `dmodapp.lib` (see Section 7.7.5 [dmodapp_lib], page 419).

Usage: inForm(I,w); I ideal or poly, w intvec

Return: ideal, generated by initial forms of generators of I w.r.t. w, or
poly, initial form of input poly w.r.t. w

Purpose: computes the initial form of an ideal or a poly w.r.t. the weight w

Note: The size of the weight vector must be equal to the number of variables of the basering.

Example:

```
LIB "dmodapp.lib";
ring r = 0,(x,y,Dx,Dy),dp;
def D = Weyl(); setring D;
poly F = 3*x^2*Dy+2*y*Dx;
poly G = 2*x*Dx+3*y*Dy+6;
ideal I = F,G;
intvec w1 = -1,-1,1,1;
intvec w2 = -1,-2,1,2;
intvec w3 = -2,-3,2,3;
inForm(I,w1);
↦ _[1]=2*y*Dx
↦ _[2]=2*x*Dx+3*y*Dy+6
inForm(I,w2);
↦ _[1]=3*x^2*Dy
↦ _[2]=2*x*Dx+3*y*Dy+6
inForm(I,w3);
↦ _[1]=3*x^2*Dy+2*y*Dx
↦ _[2]=2*x*Dx+3*y*Dy+6
inForm(F,w1);
↦ 2*y*Dx
```

7.7.5.10 restrictionIdeal

Procedure from library `dmodapp.lib` (see Section 7.7.5 [dmodapp_lib], page 419).

Usage: restrictionIdeal(I,w,[,eng,m,G]);
I ideal, w intvec, eng and m optional ints, G optional ideal

Return: ring (a Weyl algebra) containing an ideal 'resIdeal'

Assume: The basering is the n-th Weyl algebra over a field of characteristic 0 and for all 1<=i<=n the identity var(i+n)*var(i)=var(i)*var(i+1)+1 holds, i.e. the sequence of variables is given by x(1),...,x(n),D(1),...,D(n), where D(i) is the differential operator belonging to x(i).
Further, assume that I is holonomic and that w is n-dimensional with non-negative entries.

Purpose: computes the restriction ideal of a holonomic ideal to the subspace defined by the variables corresponding to the non-zero entries of the given intvec

Note: The output ring is the Weyl algebra defined by the zero entries of w. It contains an ideal 'resIdeal' being the restriction ideal of I wrt w.
If there are no zero entries, the input ring is returned.
If eng<>0, `std` is used for Groebner basis computations, otherwise, and by default, `slimgb` is used.
The minimal integer root of the b-function of I wrt the weight (-w,w) can be specified via the optional argument m.
The optional argument G is used for specifying a Groebner basis of I wrt the weight (-w,w), that is, the initial form of G generates the initial ideal of I wrt the weight (-w,w).
Further note, that the assumptions on m and G (if given) are not checked.

Display: If printlevel=1, progress debug messages will be printed,
if printlevel>=2, all the debug messages will be printed.

Example:

```
LIB "dmodapp.lib";
ring r = 0,(a,x,b,Da,Dx,Db),dp;
def D3 = Weyl();
setring D3;
ideal I = a*Db-Dx+2*Da,
x*Db-Da,
x*Da+a*Da+b*Db+1,
x*Dx-2*x*Da-a*Da,
b*Db^2+Dx*Da-Da^2+Db,
a*Dx*Da+2*x*Da^2+a*Da^2+b*Dx*Db+Dx+2*Da;
intvec w = 1,0,0;
def D2 = restrictionIdeal(I,w);
setring D2; D2;
↦ // coefficients: QQ
↦ // number of vars : 4
↦ //        block   1 : ordering C
↦ //        block   2 : ordering dp
↦ //                  : names    x b Dx Db
↦ // noncommutative relations:
↦ //    Dxx=x*Dx+1
↦ //    Dbb=b*Db+1
```

```
resIdeal;
↦ resIdeal[1]=2*x*Db-Dx
↦ resIdeal[2]=x*Dx+2*b*Db+2
↦ resIdeal[3]=4*b*Db^2+Dx^2+6*Db
```

7.7.5.11 restrictionModule

Procedure from library `dmodapp.lib` (see Section 7.7.5 [dmodapp_lib], page 419).

Usage: restrictionModule(I,w,[,eng,m,G]);
I ideal, w intvec, eng and m optional ints, G optional ideal

Return: ring (a Weyl algebra) containing a module 'resMod'

Assume: The basering is the n-th Weyl algebra over a field of characteristic 0 and for all 1<=i<=n the identity var(i+n)*var(i)=var(i)*var(i+1)+1 holds, i.e. the sequence of variables is given by x(1),...,x(n),D(1),...,D(n), where D(i) is the differential operator belonging to x(i).
Further, assume that I is holonomic and that w is n-dimensional with non-negative entries.

Purpose: computes the restriction module of a holonomic ideal to the subspace defined by the variables corresponding to the non-zero entries of the given intvec

Note: The output ring is the Weyl algebra defined by the zero entries of w.
It contains a module 'resMod' being the restriction module of I wrt w.
If there are no zero entries, the input ring is returned.
If eng<>0, `std` is used for Groebner basis computations,
otherwise, and by default, `slimgb` is used.
The minimal integer root of the b-function of I wrt the weight (-w,w) can be specified via the optional argument m.
The optional argument G is used for specifying a Groebner Basis of I wrt the weight (-w,w), that is, the initial form of G generates the initial ideal of I wrt the weight (-w,w).
Further note, that the assumptions on m and G (if given) are not checked.

Display: If printlevel=1, progress debug messages will be printed,
if printlevel>=2, all the debug messages will be printed.

Example:

```
LIB "dmodapp.lib";
ring r = 0,(a,x,b,Da,Dx,Db),dp;
def D3 = Weyl();
setring D3;
ideal I = a*Db-Dx+2*Da, x*Db-Da, x*Da+a*Da+b*Db+1,
x*Dx-2*x*Da-a*Da, b*Db^2+Dx*Da-Da^2+Db,
a*Dx*Da+2*x*Da^2+a*Da^2+b*Dx*Db+Dx+2*Da;
intvec w = 1,0,0;
def rm = restrictionModule(I,w);
setring rm; rm;
↦ // coefficients: QQ
↦ // number of vars : 4
```

```
↦ //        block   1 : ordering C
↦ //        block   2 : ordering dp
↦ //                  : names    x b Dx Db
↦ // noncommutative relations:
↦ //    Dxx=x*Dx+1
↦ //    Dbb=b*Db+1
print(resMod);
↦ 2*x*Db-Dx,x*Dx+2*b*Db+2,4*b*Db^2+Dx^2+6*Db
```

7.7.5.12 integralIdeal

Procedure from library `dmodapp.lib` (see Section 7.7.5 [dmodapp_lib], page 419).

Usage: integralIdeal(I,w,[,eng,m,G]);
I ideal, w intvec, eng and m optional ints, G optional ideal

Return: ring (a Weyl algebra) containing an ideal 'intIdeal'

Assume: The basering is the n-th Weyl algebra over a field of characteristic 0 and for all 1<=i<=n the identity var(i+n)*var(i)=var(i)*var(i+1)+1 holds, i.e. the sequence of variables is given by x(1),...,x(n),D(1),...,D(n), where D(i) is the differential operator belonging to x(i).
Further, assume that I is holonomic and that w is n-dimensional with non-negative entries.

Purpose: computes the integral ideal of a holonomic ideal w.r.t. the subspace defined by the variables corresponding to the non-zero entries of the given intvec.

Note: The output ring is the Weyl algebra defined by the zero entries of w.
It contains ideal 'intIdeal' being the integral ideal of I w.r.t. w.
If there are no zero entries, the input ring is returned.
If eng<>0, `std` is used for Groebner basis computations,
otherwise, and by default, `slimgb` is used.
The minimal integer root of the b-function of I wrt the weight (-w,w) can be specified via the optional argument m.
The optional argument G is used for specifying a Groebner basis of I wrt the weight (-w,w), that is, the initial form of G generates the initial ideal of I wrt the weight (-w,w).
Further note, that the assumptions on m and G (if given) are not checked.

Display: If printlevel=1, progress debug messages will be printed,
if printlevel>=2, all the debug messages will be printed.

Example:

```
LIB "dmodapp.lib";
ring r = 0,(x,b,Dx,Db),dp;
def D2 = Weyl();
setring D2;
ideal I = x*Dx+2*b*Db+2, x^2*Dx+b*Dx+2*x;
intvec w = 1,0;
def D1 = integralIdeal(I,w);
setring D1; D1;
```

```
↦ // coefficients: QQ
↦ // number of vars : 2
↦ //        block   1 : ordering C
↦ //        block   2 : ordering dp
↦ //                  : names    b Db
↦ // noncommutative relations:
↦ //    Dbb=b*Db+1
intIdeal;
↦ intIdeal[1]=2*b*Db+1
```

7.7.5.13 integralModule

Procedure from library `dmodapp.lib` (see Section 7.7.5 [dmodapp_lib], page 419).

Usage: integralModule(I,w,[,eng,m,G]);
I ideal, w intvec, eng and m optional ints, G optional ideal

Return: ring (a Weyl algebra) containing a module 'intMod'

Assume: The basering is the n-th Weyl algebra over a field of characteristic 0
and for all 1<=i<=n the identity var(i+n)*var(i)=var(i)*var(i+1)+1
holds, i.e. the sequence of variables is given by
x(1),...,x(n),D(1),...,D(n), where D(i) is the differential operator
belonging to x(i).
Further, assume that I is holonomic and that w is n-dimensional with
non-negative entries.

Purpose: computes the integral module of a holonomic ideal w.r.t. the subspace
defined by the variables corresponding to the non-zero entries of the
given intvec

Note: The output ring is the Weyl algebra defined by the zero entries of w.
It contains a module 'intMod' being the integral module of I wrt w.
If there are no zero entries, the input ring is returned.
If eng<>0, `std` is used for Groebner basis computations,
otherwise, and by default, `slimgb` is used.
Let F(I) denote the Fourier transform of I w.r.t. w.
The minimal integer root of the b-function of F(I) w.r.t. the weight
(-w,w) can be specified via the optional argument m.
The optional argument G is used for specifying a Groebner Basis of F(I)
wrt the weight (-w,w), that is, the initial form of G generates the
initial ideal of F(I) w.r.t. the weight (-w,w).
Further note, that the assumptions on m and G (if given) are not
checked.

Display: If printlevel=1, progress debug messages will be printed,
if printlevel>=2, all the debug messages will be printed.

Example:

```
LIB "dmodapp.lib";
ring r = 0,(x,b,Dx,Db),dp;
def D2 = Weyl();
setring D2;
ideal I = x*Dx+2*b*Db+2, x^2*Dx+b*Dx+2*x;
intvec w = 1,0;
```

```
def im = integralModule(I,w);
setring im; im;
↦ // coefficients: QQ
↦ // number of vars : 2
↦ //        block   1 : ordering C
↦ //        block   2 : ordering dp
↦ //                  : names    b Db
↦ // noncommutative relations:
↦ //    Dbb=b*Db+1
print(intMod);
↦ 2*b*Db+1,0,
↦ 0,       b*Db
```

7.7.5.14 deRhamCohom

Procedure from library `dmodapp.lib` (see Section 7.7.5 [dmodapp_lib], page 419).

Usage: deRhamCohom(f[,w,eng,m]); f poly, w optional intvec, eng and m optional ints

Return: ring (a Weyl Algebra) containing a list 'DR' of ideal and int

Assume: Basering is commutative and over a field of characteristic 0.

Purpose: computes a basis of the n-th de Rham cohomology group of the complement of the hypersurface defined by f, where n denotes the number of variables of the basering

Note: The output ring is the n-th Weyl algebra. It contains a list 'DR' with two entries (ideal J and int m) such that {f^m*J[i] : i=1..size(I)} is a basis of the n-th de Rham cohomology group of the complement of the hypersurface defined by f.
If w is an intvec with exactly n strictly positive entries, w is used in the computation. Otherwise, and by default, w is set to (1,...,1).
If eng<>0, `std` is used for Groebner basis computations, otherwise, and by default, `slimgb` is used.
If m is given, it is assumed to be less than or equal to the minimal integer root of the Bernstein-Sato polynomial of f. This assumption is not checked. If not specified, m is set to the minimal integer root of the Bernstein-Sato polynomial of f.

Theory: (SST) pp. 232-235

Display: If printlevel=1, progress debug messages will be printed, if printlevel>=2, all the debug messages will be printed.

Example:

```
LIB "dmodapp.lib";
ring r = 0,(x,y,z),dp;
poly f = x^3+y^3+z^3;
def A = deRhamCohom(f); // we see that the K-dim is 2
setring A;
DR;
↦ [1]:
↦    _[1]=-x^3*Dx*Dy*Dz
↦    _[2]=-x*y*z*Dx*Dy*Dz
```

```
↦ [2]:
↦    -2
```

See also: Section 7.7.5.15 [deRhamCohomIdeal], page 433.

7.7.5.15 deRhamCohomIdeal

Procedure from library `dmodapp.lib` (see Section 7.7.5 [dmodapp_lib], page 419).

Usage: deRhamCohomIdeal (I[,w,eng,k,G]);
I ideal, w optional intvec, eng and k optional ints, G optional ideal

Return: ideal

Assume: The basering is the n-th Weyl algebra D over a field of characteristic zero and for all 1<=i<=n the identity var(i+n)*var(i)=var(i)*var(i+1)+1 holds, i.e. the sequence of variables is given by x(1),...,x(n),D(1),...,D(n), where D(i) is the differential operator belonging to x(i).
Further, assume that I is of special kind, namely let f in K[x] and consider the module K[x,1/f]f^m, where m is smaller than or equal to the minimal integer root of the Bernstein-Sato polynomial of f.
Since this module is known to be a holonomic D-module, it has a cyclic presentation D/I.

Purpose: computes a basis of the n-th de Rham cohomology group of the complement of the hypersurface defined by f

Note: The elements of the basis are of the form f^m*p, where p runs over the entries of the returned ideal.
If I does not satisfy the assumptions described above, the result might have no meaning. Note that I can be computed with `annfs`.
If w is an intvec with exactly n strictly positive entries, w is used in the computation. Otherwise, and by default, w is set to (1,...,1).
If eng<>0, `std` is used for Groebner basis computations, otherwise, and by default, `slimgb` is used.
Let F(I) denote the Fourier transform of I wrt w.
An integer smaller than or equal to the minimal integer root of the b-function of F(I) wrt the weight (-w,w) can be specified via the optional argument k.
The optional argument G is used for specifying a Groebner Basis of F(I) wrt the weight (-w,w), that is, the initial form of G generates the initial ideal of F(I) wrt the weight (-w,w).
Further note, that the assumptions on I, k and G (if given) are not checked.

Theory: (SST) pp. 232-235

Display: If printlevel=1, progress debug messages will be printed,
if printlevel>=2, all the debug messages will be printed.

Example:

```
LIB "dmodapp.lib";
ring r = 0,(x,y,z),dp;
poly F = x^3+y^3+z^3;
bfctAnn(F);            // Bernstein-Sato poly of F has minimal integer root -2
```

```
↦ [1]:
↦    _[1]=-1
↦    _[2]=-4/3
↦    _[3]=-5/3
↦    _[4]=-2
↦ [2]:
↦    2,1,1,1
def W = annRat(1,F^2); // so we compute the annihilator of 1/F^2
setring W; W;          // Weyl algebra, contains LD = Ann(1/F^2)
↦ // coefficients: QQ
↦ // number of vars : 6
↦ //        block   1 : ordering dp
↦ //                  : names    x y z Dx Dy Dz
↦ //        block   2 : ordering C
↦ // noncommutative relations:
↦ //    Dxx=x*Dx+1
↦ //    Dyy=y*Dy+1
↦ //    Dzz=z*Dz+1
LD;                    // K[x,y,z,1/F]F^(-2) is isomorphic to W/LD as W-module
↦ LD[1]=x*Dx+y*Dy+z*Dz+6
↦ LD[2]=z^2*Dy-y^2*Dz
↦ LD[3]=z^2*Dx-x^2*Dz
↦ LD[4]=y^2*Dx-x^2*Dy
↦ LD[5]=x^3*Dz+y^3*Dz+z^3*Dz+6*z^2
↦ LD[6]=x^3*Dy+y^3*Dy+y^2*z*Dz+6*y^2
deRhamCohomIdeal(LD);  // we see that the K-dim is 2
↦ _[1]=-x^3*Dx*Dy*Dz
↦ _[2]=-x*y*z*Dx*Dy*Dz
```

See also: Section 7.7.5.14 [deRhamCohom], page 432.

7.7.5.16 charVariety

Procedure from library `dmodapp.lib` (see Section 7.7.5 [dmodapp_lib], page 419).

Usage: charVariety(I [,eng]); I an ideal, eng an optional int

Return: ring (commutative) containing an ideal 'charVar'

Purpose: computes an ideal whose zero set is the characteristic variety of I in the sense of D-module theory

Assume: The basering is the n-th Weyl algebra over a field of characteristic 0 and for all 1<=i<=n the identity var(i+n)*var(i)=var(i)*var(i+1)+1 holds, i.e. the sequence of variables is given by x(1),...,x(n),D(1),...,D(n), where D(i) is the differential operator belonging to x(i).

Note: The output ring is commutative. It contains an ideal 'charVar'.
If eng<>0, `std` is used for Groebner basis computations,
otherwise, and by default, `slimgb` is used.

Display: If `printlevel`=1, progress debug messages will be printed,
if `printlevel`>=2, all the debug messages will be printed.

Example:

```
LIB "dmodapp.lib";
ring r = 0,(x,y),Dp;
poly F = x3-y2;
printlevel = 0;
def A  = annfs(F);
setring A;      // Weyl algebra
LD;             // the annihilator of F
↦ LD[1]=2*x*Dx+3*y*Dy+6
↦ LD[2]=3*x^2*Dy+2*y*Dx
↦ LD[3]=9*x*y*Dy^2-4*y*Dx^2+15*x*Dy
↦ LD[4]=27*y^2*Dy^3+8*y*Dx^3+135*y*Dy^2+105*Dy
def CA = charVariety(LD);
setring CA; CA; // commutative ring
↦ // coefficients: QQ
↦ // number of vars : 4
↦ //        block   1 : ordering dp
↦ //                  : names    x y Dx Dy
↦ //        block   2 : ordering C
charVar;
↦ charVar[1]=2*x*Dx+3*y*Dy
↦ charVar[2]=3*x^2*Dy+2*y*Dx
↦ charVar[3]=9*x*y*Dy^2-4*y*Dx^2
↦ charVar[4]=27*y^2*Dy^3+8*y*Dx^3
dim(std(charVar));   // hence I is holonomic
↦ 2
```

See also: Section 7.7.5.17 [charInfo], page 435.

7.7.5.17 charInfo

Procedure from library `dmodapp.lib` (see Section 7.7.5 [dmodapp_lib], page 419).

Usage: charInfo(I); I an ideal

Return: ring (commut.) containing ideals 'charVar','singLoc' and list 'primDec'

Purpose: computes characteristic variety of I (in the sense of D-module theory), its singular locus and primary decomposition

Assume: The basering is the n-th Weyl algebra over a field of characteristic 0 and for all 1<=i<=n the identity var(i+n)*var(i)=var(i)*var(i+1)+1 holds, i.e. the sequence of variables is given by x(1),...,x(n),D(1),...,D(n), where D(i) is the differential operator belonging to x(i).

Note: In the output ring, which is commutative:
- the ideal 'charVar' is the characteristic variety char(I),
- the ideal 'SingLoc' is the singular locus of char(I),
- the list 'primDec' is the primary decomposition of char(I).

Display: If `printlevel`=1, progress debug messages will be printed, if `printlevel`>=2, all the debug messages will be printed.

Example:

```
LIB "dmodapp.lib";
ring r = 0,(x,y),Dp;
poly F = x3-y2;
```

```
printlevel = 0;
def A  = annfs(F);
setring A;      // Weyl algebra
LD;             // the annihilator of F
↦ LD[1]=2*x*Dx+3*y*Dy+6
↦ LD[2]=3*x^2*Dy+2*y*Dx
↦ LD[3]=9*x*y*Dy^2-4*y*Dx^2+15*x*Dy
↦ LD[4]=27*y^2*Dy^3+8*y*Dx^3+135*y*Dy^2+105*Dy
def CA = charInfo(LD);
setring CA; CA; // commutative ring
↦ // coefficients: QQ
↦ // number of vars : 4
↦ //        block   1 : ordering dp
↦ //                  : names    x y Dx Dy
↦ //        block   2 : ordering C
charVar;        // characteristic variety
↦ charVar[1]=2*x*Dx+3*y*Dy
↦ charVar[2]=3*x^2*Dy+2*y*Dx
↦ charVar[3]=9*x*y*Dy^2-4*y*Dx^2
↦ charVar[4]=27*y^2*Dy^3+8*y*Dx^3
singLoc;        // singular locus
↦ singLoc[1]=y*Dy
↦ singLoc[2]=y*Dx
↦ singLoc[3]=2*x*Dx-3*y*Dy
↦ singLoc[4]=9*x*Dy^2-2*Dx^2
↦ singLoc[5]=3*x^2*Dy-y*Dx
↦ singLoc[6]=Dx^3
↦ singLoc[7]=x^3-y^2
primDec;        // primary decomposition
↦ [1]:
↦    [1]:
↦       _[1]=Dy
↦       _[2]=Dx
↦    [2]:
↦       _[1]=Dy
↦       _[2]=Dx
↦ [2]:
↦    [1]:
↦       _[1]=27*y*Dy^3+8*Dx^3
↦       _[2]=9*x*Dy^2-4*Dx^2
↦       _[3]=2*x*Dx+3*y*Dy
↦       _[4]=3*x^2*Dy+2*y*Dx
↦       _[5]=x^3-y^2
↦    [2]:
↦       _[1]=27*y*Dy^3+8*Dx^3
↦       _[2]=9*x*Dy^2-4*Dx^2
↦       _[3]=2*x*Dx+3*y*Dy
↦       _[4]=3*x^2*Dy+2*y*Dx
↦       _[5]=x^3-y^2
↦ [3]:
↦    [1]:
↦       _[1]=y
↦       _[2]=x
```

```
↦    [2]:
↦       _[1]=y
↦       _[2]=x
```

7.7.5.18 isFsat

Procedure from library `dmodapp.lib` (see Section 7.7.5 [dmodapp_lib], page 419).

Usage: isFsat(I, F); I an ideal, F a poly

Return: int, 1 if I is F-saturated and 0 otherwise

Purpose: checks whether the ideal I is F-saturated

Note: We check indeed that Ker(D–> F–> D/I) is 0, where D is the basering.

Example:

```
LIB "dmodapp.lib";
ring r = 0,(x,y),dp;
poly G = x*(x-y)*y;
def A = annfs(G);
setring A;
poly F = x3-y2;
isFsat(LD,F);
↦ 1
ideal J = LD*F;
isFsat(J,F);
↦ 0
```

7.7.5.19 appelF1

Procedure from library `dmodapp.lib` (see Section 7.7.5 [dmodapp_lib], page 419).

Usage: appelF1();

Return: ring (a parametric Weyl algebra) containing an ideal 'IAppel1'

Purpose: defines the ideal in a parametric Weyl algebra,
which annihilates Appel F1 hypergeometric function

Note: The output ring is a parametric Weyl algebra. It contains an ideal
'IAappel1' annihilating Appel F1 hypergeometric function.
See (SST) p. 48.

Example:

```
LIB "dmodapp.lib";
def A = appelF1();
setring A;
IAppel1;
↦ IAppel1[1]=-x^3*Dx^2+x^2*Dx^2-x^2*y*Dx*Dy+x*y*Dx*Dy+(-a-b-1)*x^2*Dx+(c)*x\
   *Dx+(-b)*x*y*Dy+(-a*b)*x
↦ IAppel1[2]=-x*y^2*Dx*Dy+x*y*Dx*Dy-y^3*Dy^2+y^2*Dy^2+(-d)*x*y*Dx+(-a-d-1)*\
   y^2*Dy+(c)*y*Dy+(-a*d)*y
↦ IAppel1[3]=x*Dx*Dy-y*Dx*Dy+(-d)*Dx+(b)*Dy
```

7.7.5.20 appelF2

Procedure from library `dmodapp.lib` (see Section 7.7.5 [dmodapp_lib], page 419).

Usage: appelF2();

Return: ring (a parametric Weyl algebra) containing an ideal 'IAppel2'

Purpose: defines the ideal in a parametric Weyl algebra,
which annihilates Appel F2 hypergeometric function

Note: The output ring is a parametric Weyl algebra. It contains an ideal
'IAappel2' annihilating Appel F2 hypergeometric function.
See (SST) p. 85.

Example:

```
LIB "dmodapp.lib";
def A = appelF2();
setring A;
IAppel2;
↦ IAppel2[1]=-x^3*Dx^2+x^2*Dx^2-x^2*y*Dx*Dy+(-a-b-1)*x^2*Dx+x*Dx+(-b)*x*y*D\
   y+(-a*b)*x
↦ IAppel2[2]=-x*y^2*Dx*Dy-y^3*Dy^2+y^2*Dy^2+(-c)*x*y*Dx+(-a-c-1)*y^2*Dy+y*D\
   y+(-a*c)*y
```

7.7.5.21 appelF4

Procedure from library `dmodapp.lib` (see Section 7.7.5 [dmodapp_lib], page 419).

Usage: appelF4();

Return: ring (a parametric Weyl algebra) containing an ideal 'IAppel4'

Purpose: defines the ideal in a parametric Weyl algebra,
which annihilates Appel F4 hypergeometric function

Note: The output ring is a parametric Weyl algebra. It contains an ideal
'IAappel4' annihilating Appel F4 hypergeometric function.
See (SST) p. 39.

Example:

```
LIB "dmodapp.lib";
def A = appelF4();
setring A;
IAppel4;
↦ IAppel4[1]=-x^2*Dx^2+x*Dx^2-2*x*y*Dx*Dy-y^2*Dy^2+(-a-b-1)*x*Dx+(c)*Dx+(-a\
   -b-1)*y*Dy+(-a*b)
↦ IAppel4[2]=-x^2*Dx^2-2*x*y*Dx*Dy-y^2*Dy^2+y*Dy^2+(-a-b-1)*x*Dx+(-a-b-1)*y\
   *Dy+(d)*Dy+(-a*b)
```

7.7.5.22 fourier

Procedure from library `dmodapp.lib` (see Section 7.7.5 [dmodapp_lib], page 419).

Usage: fourier(I[,v]); I an ideal, v an optional intvec

Return: ideal

Purpose: computes the Fourier transform of an ideal in a Weyl algebra

Assume: The basering is the n-th Weyl algebra over a field of characteristic 0 and for all 1<=i<=n the identity var(i+n)*var(i)=var(i)*var(i+1)+1 holds, i.e. the sequence of variables is given by x(1),...,x(n),D(1),...,D(n), where D(i) is the differential operator belonging to x(i).

Note: The Fourier automorphism is defined by mapping x(i) to -D(i) and D(i) to x(i).
If v is an intvec with entries ranging from 1 to n, the Fourier transform of I restricted to the variables given by v is computed.

Example:

```
LIB "dmodapp.lib";
ring r = 0,(x,y,Dx,Dy),dp;
def D2 = Weyl();
setring D2;
ideal I = x*Dx+2*y*Dy+2, x^2*Dx+y*Dx+2*x;
intvec v = 2;
fourier(I,v);
↦ _[1]=x*Dx-2*y*Dy
↦ _[2]=x^2*Dx-Dx*Dy+2*x
fourier(I);
↦ _[1]=-x*Dx-2*y*Dy-1
↦ _[2]=x*Dx^2-x*Dy
```

See also: Section 7.7.5.23 [inverseFourier], page 439.

7.7.5.23 inverseFourier

Procedure from library `dmodapp.lib` (see Section 7.7.5 [dmodapp_lib], page 419).

Usage: inverseFourier(I[,v]); I an ideal, v an optional intvec

Return: ideal

Purpose: computes the inverse Fourier transform of an ideal in a Weyl algebra

Assume: The basering is the n-th Weyl algebra over a field of characteristic 0 and for all 1<=i<=n the identity var(i+n)*var(i)=var(i)*var(i+1)+1 holds, i.e. the sequence of variables is given by x(1),...,x(n),D(1),...,D(n), where D(i) is the differential operator belonging to x(i).

Note: The Fourier automorphism is defined by mapping x(i) to -D(i) and D(i) to x(i).
If v is an intvec with entries ranging from 1 to n, the inverse Fourier transform of I restricted to the variables given by v is computed.

Example:

```
LIB "dmodapp.lib";
ring r = 0,(x,y,Dx,Dy),dp;
def D2 = Weyl();
setring D2;
ideal I = x*Dx+2*y*Dy+2, x^2*Dx+y*Dx+2*x;
intvec v = 2;
ideal FI = fourier(I);
```

```
inverseFourier(FI);
↦ _[1]=x*Dx+2*y*Dy+2
↦ _[2]=x^2*Dx+y*Dx+2*x
```

See also: Section 7.7.5.22 [fourier], page 438.

7.7.5.24 bFactor

Procedure from library `dmodapp.lib` (see Section 7.7.5 [dmodapp_lib], page 419).

Usage: bFactor(f); f poly

Return: list of ideal and intvec and possibly a string

Purpose: tries to compute the roots of a univariate poly f

Note: The output list consists of two or three entries:
roots of f as an ideal, their multiplicities as intvec, and,
if present, a third one being the product of all irreducible factors
of degree greater than one, given as string.
If f is the zero polynomial or if f has no roots in the ground field,
this is encoded as root 0 with multiplicity 0.

Display: If printlevel=1, progress debug messages will be printed,
if printlevel>=2, all the debug messages will be printed.

Example:

```
LIB "dmodapp.lib";
ring r = 0,(x,y),dp;
bFactor((x^2-1)^2);
↦ [1]:
↦    _[1]=1
↦    _[2]=-1
↦ [2]:
↦    2,2
bFactor((x^2+1)^2);
↦ [1]:
↦    _[1]=0
↦ [2]:
↦    0
↦ [3]:
↦    x4+2x2+1
bFactor((y^2+1/2)*(y+9)*(y-7));
↦ [1]:
↦    _[1]=7
↦    _[2]=-9
↦ [2]:
↦    1,1
↦ [3]:
↦    2y2+1
bFactor(1);
↦ [1]:
↦    _[1]=0
↦ [2]:
↦    0
↦ [3]:
```

```
↦    1
bFactor(0);
↦ [1]:
↦    _[1]=0
↦ [2]:
↦    0
↦ [3]:
↦    0
```

7.7.5.25 intRoots

Procedure from library `dmodapp.lib` (see Section 7.7.5 [dmodapp_lib], page 419).

Usage: isInt(L); L a list

Return: list

Purpose: extracts integer roots from a list given in `bFactor` format

Assume: The input list must be given in the format of `bFactor`.

Note: Parameters are treated as integers.

Example:

```
LIB "dmodapp.lib";
ring r = 0,x,dp;
list L = bFactor((x-4/3)*(x+3)^2*(x-5)^4); L;
↦ [1]:
↦    _[1]=5
↦    _[2]=4/3
↦    _[3]=-3
↦ [2]:
↦    4,1,2
intRoots(L);
↦ [1]:
↦    _[1]=5
↦    _[2]=-3
↦ [2]:
↦    4,2
```

See also: Section 7.7.5.24 [bFactor], page 440.

7.7.5.26 poly2list

Procedure from library `dmodapp.lib` (see Section 7.7.5 [dmodapp_lib], page 419).

Usage: poly2list(f); f a poly

Return: list of exponents and corresponding terms of f

Purpose: converts a poly to a list of pairs consisting of intvecs (1st entry) and polys (2nd entry), where the i-th pair contains the exponent of the i-th term of f and the i-th term (with coefficient) itself.

Example:

```
LIB "dmodapp.lib";
ring r = 0,x,dp;
poly F = x;
```

```
poly2list(F);
↦ [1]:
↦    [1]:
↦       1
↦    [2]:
↦       x
ring r2 = 0,(x,y),dp;
poly F = x2y+5xy2;
poly2list(F);
↦ [1]:
↦    [1]:
↦       2,1
↦    [2]:
↦       x2y
↦ [2]:
↦    [1]:
↦       1,2
↦    [2]:
↦       5xy2
poly2list(0);
↦ [1]:
↦    [1]:
↦       0,0
↦    [2]:
↦       0
```

7.7.5.27 fl2poly

Procedure from library `dmodapp.lib` (see Section 7.7.5 [dmodapp_lib], page 419).

Usage: fl2poly(L,s); L a list, s a string

Return: poly

Purpose: reconstruct a monic polynomial in one variable from its factorization

Assume: s is a string with the name of some variable and
L is supposed to consist of two entries:
- L[1] of the type ideal with the roots of a polynomial
- L[2] of the type intvec with the multiplicities of corr. roots

Example:

```
LIB "dmodapp.lib";
ring r = 0,(x,y,z,s),Dp;
ideal I = -1,-4/3,0,-5/3,-2;
intvec mI = 2,1,2,1,1;
list BS = I,mI;
poly p = fl2poly(BS,"s");
p;
↦ s7+7s6+173/9s5+233/9s4+154/9s3+40/9s2
factorize(p,2);
↦ [1]:
↦    _[1]=s
↦    _[2]=s+1
↦    _[3]=3s+4
```

```
↦    _[4]=3s+5
↦    _[5]=s+2
↦ [2]:
↦    2,2,1,1,1
```

7.7.5.28 insertGenerator

Procedure from library `dmodapp.lib` (see Section 7.7.5 [dmodapp_lib], page 419).

Usage: insertGenerator(id,p[,k]);
id an ideal/module, p a poly/vector, k an optional int

Return: of the same type as id

Purpose: inserts p into id at k-th position and returns the enlarged object

Note: If k is given, p is inserted at position k, otherwise (and by default),
p is inserted at the beginning (k=1).

Example:

```
LIB "dmodapp.lib";
ring r = 0,(x,y,z),dp;
ideal I = x^2,z^4;
insertGenerator(I,y^3);
↦ _[1]=y3
↦ _[2]=x2
↦ _[3]=z4
insertGenerator(I,y^3,2);
↦ _[1]=x2
↦ _[2]=y3
↦ _[3]=z4
module M = I*gen(3);
insertGenerator(M,[x^3,y^2,z],2);
↦ _[1]=x2*gen(3)
↦ _[2]=x3*gen(1)+y2*gen(2)+z*gen(3)
↦ _[3]=z4*gen(3)
insertGenerator(M,x+y+z,4);
↦ _[1]=x2*gen(3)
↦ _[2]=z4*gen(3)
↦ _[3]=0
↦ _[4]=x*gen(1)+y*gen(1)+z*gen(1)
```

See also: Section 7.7.5.29 [deleteGenerator], page 443.

7.7.5.29 deleteGenerator

Procedure from library `dmodapp.lib` (see Section 7.7.5 [dmodapp_lib], page 419).

Usage: deleteGenerator(id,k); id an ideal/module, k an int

Return: of the same type as id

Purpose: deletes the k-th generator from the first argument and returns
the altered object

Example:

```
LIB "dmodapp.lib";
ring r = 0,(x,y,z),dp;
ideal I = x^2,y^3,z^4;
deleteGenerator(I,2);
↦ _[1]=x2
↦ _[2]=z4
module M = [x,y,z],[x2,y2,z2],[x3,y3,z3];
print(deleteGenerator(M,2));
↦ x,x3,
↦ y,y3,
↦ z,z3
M = M[1];
deleteGenerator(M,1);
↦ _[1]=0
```

See also: Section 7.7.5.28 [insertGenerator], page 443.

7.7.5.30 engine

Procedure from library `dmodapp.lib` (see Section 7.7.5 [dmodapp_lib], page 419).

Usage: engine(I,i); I ideal/module/matrix, i an int

Return: the same type as I

Purpose: compute the Groebner basis of I with the algorithm, chosen via i

Note: By default and if i=0, slimgb is used; otherwise std does the job.

Example:

```
LIB "dmodapp.lib";
ring r = 0,(x,y),Dp;
ideal I  = y*(x3-y2),x*(x3-y2);
engine(I,0); // uses slimgb
↦ _[1]=x3y-y3
↦ _[2]=x4-xy2
engine(I,1); // uses std
↦ _[1]=x3y-y3
↦ _[2]=x4-xy2
```

7.7.5.31 isInt

Procedure from library `dmodapp.lib` (see Section 7.7.5 [dmodapp_lib], page 419).

Usage: isInt(n); n a number

Return: int, 1 if n is an integer or 0 otherwise

Purpose: check whether given object of type number is actually an int

Note: Parameters are treated as integers.

Example:

```
LIB "dmodapp.lib";
ring r = 0,x,dp;
number n = 4/3;
isInt(n);
↦ 0
```

```
n = 11;
isInt(n);
↦ 1
```

7.7.5.32 sortIntvec

Procedure from library `dmodapp.lib` (see Section 7.7.5 [dmodapp_lib], page 419).

Usage: sortIntvec(v); v an intvec

Return: list of two intvecs

Purpose: sorts an intvec

Note: In the output list L, the first entry consists of the entries of v satisfying L[1][i] >= L[1][i+1]. The second entry is a permutation such that v[L[2]] = L[1].
Unlike in the procedure `sort`, zeros are not dismissed.

Example:

```
LIB "dmodapp.lib";
ring r = 0,x,dp;
intvec v = -1,0,1,-2,0,2;
list L = sortIntvec(v); L;
↦ [1]:
↦    2,1,0,0,-1,-2
↦ [2]:
↦    6,3,2,5,1,4
v[L[2]];
↦ 2 1 0 0 -1 -2
v = -3,0;
sortIntvec(v);
↦ [1]:
↦    0,-3
↦ [2]:
↦    2,1
v = 0,-3;
sortIntvec(v);
↦ [1]:
↦    0,-3
↦ [2]:
↦    1,2
```

See also: [sort], page 727.

7.7.6 dmodvar_lib

Library: dmodvar.lib

Purpose: Algebraic D-modules for varieties

Authors: Daniel Andres, daniel.andres@math.rwth-aachen.de
Viktor Levandovskyy, levandov@math.rwth-aachen.de
Jorge Martin-Morales, jorge@unizar.es

Support: DFG Graduiertenkolleg 1632 'Experimentelle und konstruktive Algebra'

Overview: Let K be a field of characteristic 0. Given a polynomial ring R = K[x_1,...,x_n] and polynomials f_1,...,f_r in R, define F = f_1*...*f_r and F^s = f_1^s_1*...*f_r^s_r for symbolic discrete (that is shiftable) variables s_1,..., s_r. The module R[1/F]*F^s has the structure of a D<S>-module, where D<S> = D(R) tensored with S over K, where
- D(R) is the n-th Weyl algebra K<x_1,...,x_n,d_1,...,d_n | d_j x_j = x_j d_j + 1>
- S is the universal enveloping algebra of gl_r, generated by s_i = s_{ii}.
One is interested in the following data:
- the left ideal Ann F^s in D<S>, usually denoted by LD in the output
- global Bernstein polynomial in one variable s = s_1+...+s_r, denoted by bs,
- its minimal integer root s0, the list of all roots of bs, which are known to be negative rational numbers, with their multiplicities, which is denoted by BS
- an r-tuple of operators in D<S>, denoted by PS, such that the functional equality sum(k=1 to k=r) P_k*f_k*F^s = bs*F^s holds in R[1/F]*F^s.

References:

(BMS06) Budur, Mustata, Saito: Bernstein-Sato polynomials of arbitrary varieties (2006).
(ALM09) Andres, Levandovskyy, Martin-Morales: Principal Intersection and Bernstein-Sato Polynomial of an Affine Variety (2009).

Procedures: See also: Section 7.7.2 [bfun_lib], page 375; Section 7.7.4 [dmod_lib], page 400; Section 7.7.5 [dmodapp_lib], page 419; Section D.6.9 [gmssing_lib], page 778.

7.7.6.1 bfctVarIn

Procedure from library `dmodvar.lib` (see Section 7.7.6 [dmodvar_lib], page 445).

Usage: bfctVarIn(I [,a,b,c]); I an ideal, a,b,c optional ints

Return: list of ideal and intvec

Purpose: computes the roots of the Bernstein-Sato polynomial and their multiplicities for an affine algebraic variety defined by I.

Assume: The basering is commutative and over a field of characteristic 0.
Varnames of the basering do not include t(1),...,t(r) and Dt(1),...,Dt(r), where r is the number of entries of the input ideal.

Note: In the output list, say L,
- L[1] of type ideal contains all the rational roots of a b-function,
- L[2] of type intvec contains the multiplicities of above roots,
- optional L[3] of type string is the part of b-function without rational roots.
Note, that a b-function of degree 0 is encoded via L[1][1]=0, L[2]=0 and L[3] is 1 (for nonzero constant) or 0 (for zero b-function).
If a<>0, the ideal is used as given. Otherwise, and by default, a heuristically better suited generating set is used to reduce computation time.
If b<>0, `std` is used for GB computations in characteristic 0, otherwise, and by default, `slimgb` is used.
If c<>0, a matrix ordering is used for GB computations, otherwise, and by default, a block ordering is used.
Further note, that in this proc, the initial ideal of the multivariate Malgrange ideal defined by I is computed and then a system of linear equations is solved by linear reductions following the ideas by Noro.
The result is shifted by 1-codim(Var(F)) following (BMS06).

Display: If printlevel=1, progress debug messages will be printed,
if printlevel>=2, all the debug messages will be printed.

Example:

```
LIB "dmodvar.lib";
ring R = 0,(x,y,z),dp;
ideal F = x^2+y^3, z;
list L = bfctVarIn(F);
L;
↦ [1]:
↦    _[1]=-5/6
↦    _[2]=-1
↦    _[3]=-7/6
↦ [2]:
↦    1,1,1
```

7.7.6.2 bfctVarAnn

Procedure from library `dmodvar.lib` (see Section 7.7.6 [dmodvar_lib], page 445).

Usage: bfctVarAnn(F[,gid,eng]); F an ideal, gid,eng optional ints

Return: list of an ideal and an intvec

Purpose: computes the roots of the Bernstein-Sato polynomial and their multiplicities for an affine algebraic variety defined by F = F[1],..,F[r].

Assume: The basering is commutative and over a field in char 0.

Note: In the output list, the ideal contains all the roots and the intvec their multiplicities.
If gid<>0, the ideal is used as given. Otherwise, and by default, a heuristically better suited generating set is used.
If eng<>0, `std` is used for GB computations, otherwise, and by default, `slimgb` is used.
Computational remark: The time of computation can be very different depending on the chosen generators of F, although the result is always the same.
Further note that in this proc, the annihilator of f^s in D[s] is computed and then a system of linear equations is solved by linear reductions in order to find the minimal polynomial of S = s(1)(1) + ... + s(P)(P). The resulted is shifted by 1-codim(Var(F)) following (BMS06).

Display: If printlevel=1, progress debug messages will be printed,
if printlevel=2, all the debug messages will be printed.

Example:

```
LIB "dmodvar.lib";
ring R = 0,(x,y,z),Dp;
ideal F = x^2+y^3, z;
bfctVarAnn(F);
↦ [1]:
↦    _[1]=-5/6
↦    _[2]=-1
↦    _[3]=-7/6
↦ [2]:
↦    1,1,1
```

7.7.6.3 SannfsVar

Procedure from library `dmodvar.lib` (see Section 7.7.6 [dmodvar_lib], page 445).

Usage: SannfsVar(F [,ORD,eng]); F an ideal, ORD an optional string, eng an optional int

Return: ring (Weyl algebra tensored with U(gl_P)), containing an ideal LD

Purpose: compute the D<S>-module structure of D<S>*f^s where f = F[1]*...*F[P] and D<S> is the Weyl algebra D tensored with K<S>=U(gl_P), according to the generalized algorithm by Briancon and Maisonobe for affine varieties

Assume: The basering is commutative and over a field of characteristic 0.

Note: Activate the output ring D<S> with the `setring` command. In the ring D<S>, the ideal LD is the needed D<S>-module structure.
The value of ORD must be an elimination ordering in D<Dt,S> for Dt written in the string form, otherwise the result may have no meaning. By default ORD = '(a(1..(P)..1),a(1..(P+P^2)..1),dp)'.
If eng<>0, `std` is used for Groebner basis computations, otherwise, and by default `slimgb` is used.

Display: If printlevel=1, progress debug messages will be printed,
if printlevel>=2, all the debug messages will be printed.

Example:

```
LIB "dmodvar.lib";
ring R = 0,(x,y),Dp;
ideal F = x^3, y^5;
//ORD = "(a(1,1),a(1,1,1,1,1,1),dp)";
//eng = 0;
def A = SannfsVar(F);
setring A;
A;
↦ // coefficients: QQ
↦ // number of vars : 8
↦ //        block   1 : ordering a
↦ //                  : names    s(1)(1) s(1)(2) s(2)(1) s(2)(2)
↦ //                  : weights        1       1       1       1
↦ //        block   2 : ordering dp
↦ //                  : names    s(1)(1) s(1)(2) s(2)(1) s(2)(2) x y Dx Dy
↦ //        block   3 : ordering C
↦ // noncommutative relations:
↦ //    s(1)(2)s(1)(1)=s(1)(1)*s(1)(2)-s(1)(2)
↦ //    s(2)(1)s(1)(1)=s(1)(1)*s(2)(1)+s(2)(1)
↦ //    s(2)(1)s(1)(2)=s(1)(2)*s(2)(1)-s(1)(1)+s(2)(2)
↦ //    s(2)(2)s(1)(2)=s(1)(2)*s(2)(2)-s(1)(2)
↦ //    s(2)(2)s(2)(1)=s(2)(1)*s(2)(2)+s(2)(1)
↦ //    Dxx=x*Dx+1
↦ //    Dyy=y*Dy+1
LD;
↦ LD[1]=5*s(2)(2)-y*Dy
↦ LD[2]=3*s(1)(1)-x*Dx
↦ LD[3]=15*s(1)(2)*s(2)(1)-x*y*Dx*Dy-5*x*Dx
↦ LD[4]=5*s(2)(1)*y^4-x^3*Dy
↦ LD[5]=3*s(1)(2)*x^2-y^5*Dx
```

7.7.6.4 makeMalgrange

Procedure from library `dmodvar.lib` (see Section 7.7.6 [dmodvar_lib], page 445).

Usage: makeMalgrange(F [,ORD]); F an ideal, ORD an optional string

Return: ring (Weyl algebra) containing an ideal IF

Purpose: create the ideal by Malgrange associated with F = F[1],...,F[P].

Note: Activate the output ring with the `setring` command. In this ring, the ideal IF is the ideal by Malgrange corresponding to F.
The value of ORD must be an arbitrary ordering in K<_t,_x,_Dt,_Dx> written in the string form. By default ORD = 'dp'.

Display: If printlevel=1, progress debug messages will be printed,
if printlevel>=2, all the debug messages will be printed.

Example:

```
LIB "dmodvar.lib";
ring R = 0,(x,y,z),Dp;
ideal I = x^2+y^3, z;
def W = makeMalgrange(I);
setring W;
W;
↦ // coefficients: QQ
↦ // number of vars : 10
↦ //        block   1 : ordering dp
↦ //                  : names    t(1) t(2) x y z Dt(1) Dt(2) Dx Dy Dz
↦ //        block   2 : ordering C
↦ // noncommutative relations:
↦ //    Dt(1)t(1)=t(1)*Dt(1)+1
↦ //    Dt(2)t(2)=t(2)*Dt(2)+1
↦ //    Dxx=x*Dx+1
↦ //    Dyy=y*Dy+1
↦ //    Dzz=z*Dz+1
IF;
↦ IF[1]=-y^3-x^2+t(1)
↦ IF[2]=t(2)-z
↦ IF[3]=2*x*Dt(1)+Dx
↦ IF[4]=3*y^2*Dt(1)+Dy
↦ IF[5]=Dt(2)+Dz
```

7.7.7 fpadim_lib

Library: fpadim.lib

Purpose: Algorithms for quotient algebras in the letterplace case

Authors: Grischa Studzinski, grischa.studzinski at rwth-aachen.de
Viktor Levandovskyy, viktor.levandovskyy at math.rwth-aachen.de
Karim Abou Zeid, karim.abou.zeid at rwth-aachen.de

Support: Joint projects LE 2697/2-1 and KR 1907/3-1 of the Priority Programme SPP 1489: 'Algorithmische und Experimentelle Methoden in Algebra, Geometrie und Zahlentheorie' of the German DFG
and Project II.6 of the transregional collaborative research centre SFB-TRR 195 'Symbolic Tools in Mathematics and their Application' of the German DFG

Overview: Given the free associative algebra A = K<x_1,...,x_n> and a (finite) Groebner basis GB = {g_1,..,g_w}, one is interested in the K-dimension and in the explicit K-basis of A/<GB>.
Therefore one is interested in the following data:
- the Ufnarovskij graph induced by GB
- the mistletoes of A/<GB> (special monomials in a basis) - the K-dimension of A/<GB>
- the Hilbert series of A/<GB>

The Ufnarovskij graph is used to determine whether A/<GB> has finite K-dimension. One has to check if the graph contains cycles. For the whole theory we refer to [ufna]. Given a reduced set of monomials GB one can define the basis tree, whose vertex set V consists of all normal monomials w.r.t. GB. For every two monomials m_1, m_2 in V there is a direct edge from m_1 to m_2, if and only if there exists x_k in {x_1,..,x_n}, such that m_1*x_k = m_2. The set M = {m in V | there is no edge from m to another monomial in V} is called the set of mistletoes. As one can easily see it consists of the endpoints of the graph. Since there is a unique path to every monomial in V the whole graph can be described only from the knowledge of the mistletoes. Note that V corresponds to a basis of A/<GB>, so knowing the mistletoes we know a K-basis. The name mistletoes was given to those points because of these miraculous value and the algorithm is named sickle, because a sickle is the tool to harvest mistletoes. For more details see [studzins]. This package uses the Letterplace format introduced by [lls]. The algebra can either be represented as a Letterplace ring or via integer vectors: Every variable will only be represented by its number, so variable one is represented as 1, variable two as 2 and so on. The monomial x_1*x_3*x_2 for example will be stored as (1,3,2). Multiplication is concatenation. Note that the approach in this library does not need an algorithm for computing the normal form yet. Note that the name fpadim.lib is short for dimensions of finite presented algebras.

References:

[ufna] Ufnarovskij: Combinatorical and asymptotic methods in algebra, 1990
[lls] Levandovskyy, La Scala: Letterplace ideals and non-commutative Groebner bases, 2009
[studzins] Studzinski: Dimension computations in non-commutative, associative algebras, Diploma thesis, RWTH Aachen, 2010

Note: - basering is always a Letterplace ring
- all intvecs correspond to Letterplace monomials
- if you specify a different degree bound d, d <= attrib(basering,uptodeg) holds

In the procedures below, 'iv' stands for intvec representation and 'lp' for the letterplace representation of monomials

Procedures: See also: Section 7.7.9 [freegb_lib], page 495.

7.7.7.1 lpMis2Dim

Procedure from library `fpadim.lib` (see Section 7.7.7 [fpadim_lib], page 449).

Usage: lpMis2Dim(M); M an ideal

Return: int, the K-dimension of the factor algebra

Purpose: Computing the K-dimension out of given mistletoes

Assume: - basering is a Letterplace ring.
- M contains only monomials

Note: - The mistletoes have to be ordered lexicographically -> OrdMisLex.

Example:

```
LIB "fpadim.lib";
ring r = 0,(x,y),dp;
def R = makeLetterplaceRing(5); // constructs a Letterplace ring
setring R; // sets basering to Letterplace ring
ideal L = x(1)*y(2),y(1)*x(2)*y(3);
// ideal containing the mistletoes
lpMis2Dim(L); // returns the K-dimension of the factor algebra
↦ 6
```

7.7.7.2 lpKDim

Procedure from library `fpadim.lib` (see Section 7.7.7 [fpadim_lib], page 449).

Usage: lpKDim(G[,degbound, n]); G an ideal, degbound, n optional integers

Return: int, the K-dimension of the factor algebra

Purpose: Computing the K-dimension of a factor algebra, given via an ideal

Assume: - basering is a Letterplace ring
- if you specify a different degree bound degbound,
degbound <= attrib(basering,uptodeg) holds.

Note: - If degbound is set, there will be a degree bound added. 0 means no
degree bound. Default: attrib(basering, uptodeg).
- n is the number of variables, which can be set to a different number.
Default: attrib(basering, lV).
- If the K-dimension is known to be infinite, a degree bound is needed

Example:

```
LIB "fpadim.lib";
ring r = 0,(x,y),dp;
def R = makeLetterplaceRing(5); // constructs a Letterplace ring
setring R; // sets basering to Letterplace ring
ideal G = x(1)*x(2), y(1)*y(2),x(1)*y(2)*x(3);
// ideal G contains a Groebner basis
lpKDim(G); //procedure invoked with ring parameters
↦ 6
// the factor algebra is finite, so the degree bound given by the Letterplace
// ring is not necessary
lpKDim(G,0); // procedure without any degree bound
↦ 6
```

7.7.7.3 lpDimCheck

Procedure from library `fpadim.lib` (see Section 7.7.7 [fpadim_lib], page 449).

Usage: lpDimCheck(G);

Return: int, 1 if K-dimension of the factor algebra is infinite, 0 otherwise

Purpose: Checking a factor algebra for finiteness of the K-dimension

Assume: - basering is a Letterplace ring.

Example:

```
LIB "fpadim.lib";
ring r = 0,(x,y),dp;
def R = makeLetterplaceRing(5); // constructs a Letterplace ring
setring R; // sets basering to Letterplace ring
ideal G = x(1)*x(2), y(1)*y(2),x(1)*y(2)*x(3);
// Groebner basis
ideal I = x(1)*x(2), y(1)*x(2)*y(3), x(1)*y(2)*x(3);
// Groebner basis
lpDimCheck(G); // invokes procedure, factor algebra is of finite K-dimension
↦ 0
lpDimCheck(I); // invokes procedure, factor algebra is of infinite Kdimension
↦ 1
```

7.7.7.4 lpMis2Base

Procedure from library `fpadim.lib` (see Section 7.7.7 [fpadim_lib], page 449).

Usage: lpMis2Base(M); M an ideal

Return: ideal, a K-basis of the factor algebra

Purpose: Computing a K-basis out of given mistletoes

Assume: - basering is a Letterplace ring. G is a Letterplace ideal.
- M contains only monomials

Note: - The mistletoes have to be ordered lexicographically -> OrdMisLex.

Example:

```
LIB "fpadim.lib";
ring r = 0,(x,y),dp;
def R = makeLetterplaceRing(5); // constructs a Letterplace ring
setring R; // sets basering to Letterplace ring
ideal L = x(1)*y(2),y(1)*x(2)*y(3);
// ideal containing the mistletoes
lpMis2Base(L); // returns the K-basis of the factor algebra
↦ _[1]=y(1)
↦ _[2]=y(1)*x(2)
↦ _[3]=y(1)*x(2)*y(3)
↦ _[4]=x(1)
↦ _[5]=x(1)*y(2)
↦ _[6]=1
```

7.7.7.5 lpHilbert

Procedure from library `fpadim.lib` (see Section 7.7.7 [fpadim_lib], page 449).

Usage: lpHilbert(G[,degbound,n]); G an ideal, degbound, n optional integers

Return: intvec, containing the coefficients of the Hilbert series

Purpose: Computing the Hilbert series

Assume: - basering is a Letterplace ring.
- if you specify a different degree bound degbound,
degbound <= attrib(basering,uptodeg) holds.

Note: - If degbound is set, there will be a degree bound added. 0 means no degree bound. Default: attrib(basering,uptodeg).
- n is the number of variables, which can be set to a different number.
Default: attrib(basering, lV).
- If I is returned, then I[k] is the (k-1)-th coefficient of the Hilbert series.
- If the K-dimension is known to be infinite, a degree bound is needed

Example:

```
LIB "fpadim.lib";
ring r = 0,(x,y),dp;
def R = makeLetterplaceRing(5); // constructs a Letterplace ring
setring R; // sets basering to Letterplace ring
ideal G = x(1)*x(2), y(1)*y(2),x(1)*y(2)*x(3); // ideal G contains a
//Groebner basis
lpHilbert(G,5,2); // invokes procedure with degree bound 5 and 2 variables
↦ 1,2,2,1
// note that the optional parameters are not necessary, due to the finiteness
// of the K-dimension of the factor algebra
lpDHilbert(G); // procedure with ring parameters
↦ [1]:
↦    6
↦ [2]:
↦    1,2,2,1
lpDHilbert(G,0); // procedure without degreebound
↦ [1]:
↦    6
↦ [2]:
↦    1,2,2,1
```

7.7.7.6 lpDHilbert

Procedure from library `fpadim.lib` (see Section 7.7.7 [fpadim_lib], page 449).

Usage: lpDHilbert(G[,degbound,n]); G an ideal, degbound, n optional integers

Return: list

Purpose: Computing K-dimension and Hilbert series, starting with a lp-ideal

Assume: - basering is a Letterplace ring.
- if you specify a different degree bound degbound,
degbound <= attrib(basering,uptodeg) holds.

Note: - If L is the list returned, then L[1] is an integer corresponding to the dimension, L[2] is an intvec which contains the coefficients of the Hilbert series
- If degbound is set, there will be a degree bound added. 0 means no degree bound. Default: attrib(basering,uptodeg).
- n can be set to a different number of variables.
Default: n = attrib(basering, lV).
- If I = L[2] is the intvec returned, then I[k] is the (k-1)-th coefficient of the Hilbert series.
- If the K-dimension is known to be infinite, a degree bound is needed

Example:

```
LIB "fpadim.lib";
ring r = 0,(x,y),dp;
def R = makeLetterplaceRing(5); // constructs a Letterplace ring
setring R; // sets basering to Letterplace ring
ideal G = x(1)*x(2), y(1)*y(2),x(1)*y(2)*x(3); // ideal G contains a
//Groebner basis
lpDHilbert(G,5,2); // invokes procedure with degree bound 5 and 2 variables
↦ [1]:
↦    6
↦ [2]:
↦    1,2,2,1
// note that the optional parameters are not necessary, due to the finiteness
// of the K-dimension of the factor algebra
lpDHilbert(G); // procedure with ring parameters
↦ [1]:
↦    6
↦ [2]:
↦    1,2,2,1
lpDHilbert(G,0); // procedure without degreebound
↦ [1]:
↦    6
↦ [2]:
↦    1,2,2,1
```

7.7.7.7 lpDHilbertSickle

Procedure from library `fpadim.lib` (see Section 7.7.7 [fpadim_lib], page 449).

Usage: lpDHilbertSickle(G[,degbound,n]); G an ideal, degbound, n optional integers

Return: list

Purpose: Computing K-dimension, Hilbert series and mistletoes at once

Assume: - basering is a Letterplace ring.
- if you specify a different degree bound degbound, degbound <= attrib(basering,uptodeg) holds.

Note: - If L is the list returned, then L[1] is an integer, the K-dimension, L[2] is an intvec, the Hilbert series and L[3] is an ideal, the mistletoes
- If degbound is set, there will be a degree bound added. 0 means no

degree bound. Default: attrib(basering,uptodeg).
- n can be set to a different number of variables.
Default: n = attrib(basering, lV).
- If I = L[1] is the intvec returned, then I[k] is the (k-1)-th
coefficient of the Hilbert series.
- If the K-dimension is known to be infinite, a degree bound is needed

Example:

```
LIB "fpadim.lib";
ring r = 0,(x,y),dp;
def R = makeLetterplaceRing(5); // constructs a Letterplace ring
setring R; // sets basering to Letterplace ring
ideal G = x(1)*x(2), y(1)*y(2),x(1)*y(2)*x(3); // ideal G contains a
//Groebner basis
lpDHilbertSickle(G,5,2); //invokes procedure with degree bound 5 and 2 variables
↦ [1]:
↦    6
↦ [2]:
↦    1,2,2,1
↦ [3]:
↦    _[1]=x(1)*y(2)
↦    _[2]=y(1)*x(2)*y(3)
// note that the optional parameters are not necessary, due to the finiteness
// of the K-dimension of the factor algebra
lpDHilbertSickle(G); // procedure with ring parameters
↦ [1]:
↦    6
↦ [2]:
↦    1,2,2,1
↦ [3]:
↦    _[1]=x(1)*y(2)
↦    _[2]=y(1)*x(2)*y(3)
lpDHilbertSickle(G,0); // procedure without degreebound
↦ [1]:
↦    6
↦ [2]:
↦    1,2,2,1
↦ [3]:
↦    _[1]=x(1)*y(2)
↦    _[2]=y(1)*x(2)*y(3)
```

7.7.7.8 ivDHilbert

Procedure from library `fpadim.lib` (see Section 7.7.7 [fpadim_lib], page 449).

Usage: ivDHilbert(L,n[,degbound]); L a list of intmats, n an integer,
degbound an optional integer

Return: list

Purpose: Computing the K-dimension and the Hilbert series

Assume: - basering is a Letterplace ring
- all rows of each intmat correspond to a Letterplace monomial
- if you specify a different degree bound degbound,
degbound <= attrib(basering,uptodeg) holds

Note: - If L is the list returned, then L[1] is an integer corresponding to the dimension, L[2] is an intvec which contains the coefficients of the Hilbert series
- If degbound is set, there will be a degree bound added. By default there is no degree bound
- n is the number of variables
- If I = L[2] is the intvec returned, then I[k] is the (k-1)-th coefficient of the Hilbert series.
- If the K-dimension is known to be infinite, a degree bound is needed

Example:

```
LIB "fpadim.lib";
ring r = 0,(x,y),dp;
def R = makeLetterplaceRing(5); // constructs a Letterplace ring
R;
↦ // coefficients: QQ
↦ // number of vars : 10
↦ //        block   1 : ordering a
↦ //                  : names    x(1) y(1) x(2) y(2) x(3) y(3) x(4) y(4) x(\
   5) y(5)
↦ //                  : weights     1    1    1    1    1    1    1    1   \
    1    1
↦ //        block   2 : ordering dp
↦ //                  : names    x(1) y(1)
↦ //        block   3 : ordering dp
↦ //                  : names    x(2) y(2)
↦ //        block   4 : ordering dp
↦ //                  : names    x(3) y(3)
↦ //        block   5 : ordering dp
↦ //                  : names    x(4) y(4)
↦ //        block   6 : ordering dp
↦ //                  : names    x(5) y(5)
↦ //        block   7 : ordering C
setring R; // sets basering to Letterplace ring
//some intmats, which contain monomials in intvec representation as rows
intmat I1 [2][2] = 1,1,2,2; intmat I2 [1][3]  = 1,2,1;
intmat J1 [1][2] =  1,1; intmat J2 [2][3] = 2,1,2,1,2,1;
print(I1);
↦      1     1
↦      2     2
print(I2);
↦      1     2     1
print(J1);
↦      1     1
print(J2);
↦      2     1     2
↦      1     2     1
list G = I1,I2; // ideal, which is already a Groebner basis
list I = J1,J2; // ideal, which is already a Groebner basis
//the procedure without a degree bound
ivDHilbert(G,2);
↦ [1]:
↦    6
```

```
↦ [2]:
↦    1,2,2,1
// the procedure with degree bound 5
ivDHilbert(I,2,5);
↦ [1]:
↦    17
↦ [2]:
↦    1,2,3,3,4,4
```

7.7.7.9 ivDHilbertSickle

Procedure from library `fpadim.lib` (see Section 7.7.7 [fpadim_lib], page 449).

Usage: ivDHilbertSickle(L,n[,degbound]); L a list of intmats, n an integer, degbound an optional integer

Return: list

Purpose: Computing K-dimension, Hilbert series and mistletoes

Assume: - basering is a Letterplace ring.
- All rows of each intmat correspond to a Letterplace monomial.
- If you specify a different degree bound degbound, degbound <= attrib(basering,uptodeg) holds.

Note: - If L is the list returned, then L[1] is an integer, L[2] is an intvec which contains the coefficients of the Hilbert series and L[3] is a list, containing the mistletoes as intvecs.
- If degbound is set, a degree bound will be added. By default there is no degree bound.
- n is the number of variables.
- If I = L[2] is the intvec returned, then I[k] is the (k-1)-th coefficient of the Hilbert series.
- If the K-dimension is known to be infinite, a degree bound is needed

Example:

```
LIB "fpadim.lib";
ring r = 0,(x,y),dp;
def R = makeLetterplaceRing(5); // constructs a Letterplace ring
R;
↦ // coefficients: QQ
↦ // number of vars : 10
↦ //        block   1 : ordering a
↦ //                  : names    x(1) y(1) x(2) y(2) x(3) y(3) x(4) y(4) x(\
   5) y(5)
↦ //                  : weights     1    1    1    1    1    1    1    1   \
    1    1
↦ //        block   2 : ordering dp
↦ //                  : names    x(1) y(1)
↦ //        block   3 : ordering dp
↦ //                  : names    x(2) y(2)
↦ //        block   4 : ordering dp
↦ //                  : names    x(3) y(3)
↦ //        block   5 : ordering dp
↦ //                  : names    x(4) y(4)
```

```
↦ //        block   6 : ordering dp
↦ //                  : names    x(5) y(5)
↦ //        block   7 : ordering C
setring R; // sets basering to Letterplace ring
//some intmats, which contain monomials in intvec representation as rows
intmat I1 [2][2] = 1,1,2,2; intmat I2 [1][3]  = 1,2,1;
intmat J1 [1][2] =  1,1; intmat J2 [2][3] = 2,1,2,1,2,1;
print(I1);
↦      1     1
↦      2     2
print(I2);
↦      1     2     1
print(J1);
↦      1     1
print(J2);
↦      2     1     2
↦      1     2     1
list G = I1,I2;// ideal, which is already a Groebner basis
list I = J1,J2; // ideal, which is already a Groebner basis
ivDHilbertSickle(G,2); // invokes the procedure without a degree bound
↦ [1]:
↦    6
↦ [2]:
↦    1,2,2,1
↦ [3]:
↦    [1]:
↦       1,2
↦    [2]:
↦       2,1,2
ivDHilbertSickle(I,2,3); // invokes the procedure with degree bound 3
↦ [1]:
↦    9
↦ [2]:
↦    1,2,3,3
↦ [3]:
↦    [1]:
↦       1,2,2
↦    [2]:
↦       2,1
↦    [3]:
↦       2,2,1
↦    [4]:
↦       2,2,2
```

7.7.7.10 ivDimCheck

Procedure from library `fpadim.lib` (see Section 7.7.7 [fpadim_lib], page 449).

Usage: ivDimCheck(L,n); L a list of intmats, n an integer

Return: int, 0 if the dimension is finite, or 1 otherwise

Purpose: Decides, whether the K-dimension is finite or not

Assume: - basering is a Letterplace ring.
- All rows of each intmat correspond to a Letterplace monomial.

Note: - n is the number of variables.

Example:

```
LIB "fpadim.lib";
ring r = 0,(x,y),dp;
def R = makeLetterplaceRing(5); // constructs a Letterplace ring
R;
↦ // coefficients: QQ
↦ // number of vars : 10
↦ //        block   1 : ordering a
↦ //                  : names    x(1) y(1) x(2) y(2) x(3) y(3) x(4) y(4) x(\
   5) y(5)
↦ //                  : weights     1    1    1    1    1    1    1    1   \
    1    1
↦ //        block   2 : ordering dp
↦ //                  : names    x(1) y(1)
↦ //        block   3 : ordering dp
↦ //                  : names    x(2) y(2)
↦ //        block   4 : ordering dp
↦ //                  : names    x(3) y(3)
↦ //        block   5 : ordering dp
↦ //                  : names    x(4) y(4)
↦ //        block   6 : ordering dp
↦ //                  : names    x(5) y(5)
↦ //        block   7 : ordering C
setring R; // sets basering to Letterplace ring
//some intmats, which contain monomials in intvec representation as rows
intmat I1 [2][2] = 1,1,2,2; intmat I2 [1][3]  = 1,2,1;
intmat J1 [1][2] =  1,1; intmat J2 [2][3] = 2,1,2,1,2,1;
print(I1);
↦      1     1
↦      2     2
print(I2);
↦      1     2     1
print(J1);
↦      1     1
print(J2);
↦      2     1     2
↦      1     2     1
list G = I1,I2;// ideal, which is already a Groebner basis
list I = J1,J2; // ideal, which is already a Groebner basis and which
ivDimCheck(G,2); // invokes the procedure, factor is of finite K-dimension
↦ 0
ivDimCheck(I,2); // invokes the procedure, factor is not of finite K-dimension
↦ 1
```

7.7.7.11 ivHilbert

Procedure from library `fpadim.lib` (see Section 7.7.7 [fpadim_lib], page 449).

Usage: ivHilbert(L,n[,degbound]); L a list of intmats, n an integer, degbound an optional integer

Return: intvec, containing the coefficients of the Hilbert series

Purpose: Computing the Hilbert series

Assume: - basering is a Letterplace ring.
- all rows of each intmat correspond to a Letterplace monomial
- if you specify a different degree bound degbound,
degbound <= attrib(basering,uptodeg) holds.

Note: - If degbound is set, a degree bound will be added. By default there is no degree bound.
- n is the number of variables.
- If I is returned, then I[k] is the (k-1)-th coefficient of the Hilbert series.
- If the K-dimension is known to be infinite, a degree bound is needed

Example:

```
LIB "fpadim.lib";
ring r = 0,(x,y),dp;
def R = makeLetterplaceRing(5); // constructs a Letterplace ring
R;
↦ // coefficients: QQ
↦ // number of vars : 10
↦ //        block   1 : ordering a
↦ //                  : names    x(1) y(1) x(2) y(2) x(3) y(3) x(4) y(4) x(\
   5) y(5)
↦ //                  : weights     1    1    1    1    1    1    1    1   \
    1    1
↦ //        block   2 : ordering dp
↦ //                  : names    x(1) y(1)
↦ //        block   3 : ordering dp
↦ //                  : names    x(2) y(2)
↦ //        block   4 : ordering dp
↦ //                  : names    x(3) y(3)
↦ //        block   5 : ordering dp
↦ //                  : names    x(4) y(4)
↦ //        block   6 : ordering dp
↦ //                  : names    x(5) y(5)
↦ //        block   7 : ordering C
setring R; // sets basering to Letterplace ring
//some intmats, which contain monomials in intvec representation as rows
intmat I1 [2][2] = 1,1,2,2; intmat I2 [1][3]  = 1,2,1;
intmat J1 [1][2] =  1,1; intmat J2 [2][3] = 2,1,2,1,2,1;
print(I1);
↦      1     1
↦      2     2
print(I2);
↦      1     2     1
print(J1);
↦      1     1
print(J2);
↦      2     1     2
↦      1     2     1
list G = I1,I2; // ideal, which is already a Groebner basis
list I = J1,J2; // ideal, which is already a Groebner basis
ivHilbert(G,2); // invokes the procedure without any degree bound
```

```
↦ 1,2,2,1
ivHilbert(I,2,5); // invokes the procedure with degree bound 5
↦ 1,2,3,3,4,4
```

7.7.7.12 ivKDim

Procedure from library `fpadim.lib` (see Section 7.7.7 [fpadim_lib], page 449).

Usage: ivKDim(L,n[,degbound]); L a list of intmats,
n an integer, degbound an optional integer

Return: int, the K-dimension of A/<L>

Purpose: Computing the K-dimension of A/<L>

Assume: - basering is a Letterplace ring.
- all rows of each intmat correspond to a Letterplace monomial
- if you specify a different degree bound degbound,
degbound <= attrib(basering,uptodeg) holds.

Note: - If degbound is set, a degree bound will be added. By default there
is no degree bound.
- n is the number of variables.
- If the K-dimension is known to be infinite, a degree bound is needed

Example:

```
LIB "fpadim.lib";
ring r = 0,(x,y),dp;
def R = makeLetterplaceRing(5); // constructs a Letterplace ring
R;
↦ // coefficients: QQ
↦ // number of vars : 10
↦ //        block   1 : ordering a
↦ //                  : names    x(1) y(1) x(2) y(2) x(3) y(3) x(4) y(4) x(\
   5) y(5)
↦ //                  : weights     1    1    1    1    1    1    1    1   \
    1    1
↦ //        block   2 : ordering dp
↦ //                  : names    x(1) y(1)
↦ //        block   3 : ordering dp
↦ //                  : names    x(2) y(2)
↦ //        block   4 : ordering dp
↦ //                  : names    x(3) y(3)
↦ //        block   5 : ordering dp
↦ //                  : names    x(4) y(4)
↦ //        block   6 : ordering dp
↦ //                  : names    x(5) y(5)
↦ //        block   7 : ordering C
setring R; // sets basering to Letterplace ring
//some intmats, which contain monomials in intvec representation as rows
intmat I1 [2][2] = 1,1,2,2; intmat I2 [1][3]  = 1,2,1;
intmat J1 [1][2] =  1,1; intmat J2 [2][3] = 2,1,2,1,2,1;
print(I1);
↦      1     1
↦      2     2
```

```
print(I2);
↦      1     2     1
print(J1);
↦      1     1
print(J2);
↦      2     1     2
↦      1     2     1
list G = I1,I2; // ideal, which is already a Groebner basis
list I = J1,J2; // ideal, which is already a Groebner basis
ivKDim(G,2); // invokes the procedure without any degree bound
↦ 6
ivKDim(I,2,5); // invokes the procedure with degree bound 5
↦ 17
```

7.7.7.13 ivMis2Base

Procedure from library `fpadim.lib` (see Section 7.7.7 [fpadim_lib], page 449).

Usage: ivMis2Base(M); M a list of intvecs

Return: ideal, a K-base of the given algebra

Purpose: Computing the K-base out of given mistletoes

Assume: - The mistletoes have to be ordered lexicographically -> OrdMisLex.
Otherwise there might some elements missing.
- basering is a Letterplace ring.
- mistletoes are stored as intvecs, as described in the overview

Example:

```
LIB "fpadim.lib";
ring r = 0,(x,y),dp;
def R = makeLetterplaceRing(5); // constructs a Letterplace ring
R;
↦ // coefficients: QQ
↦ // number of vars : 10
↦ //        block   1 : ordering a
↦ //                  : names    x(1) y(1) x(2) y(2) x(3) y(3) x(4) y(4) x(\
   5) y(5)
↦ //                  : weights     1    1    1    1    1    1    1    1   \
    1    1
↦ //        block   2 : ordering dp
↦ //                  : names    x(1) y(1)
↦ //        block   3 : ordering dp
↦ //                  : names    x(2) y(2)
↦ //        block   4 : ordering dp
↦ //                  : names    x(3) y(3)
↦ //        block   5 : ordering dp
↦ //                  : names    x(4) y(4)
↦ //        block   6 : ordering dp
↦ //                  : names    x(5) y(5)
↦ //        block   7 : ordering C
setring R; // sets basering to Letterplace ring
intvec i1 = 1,2; intvec i2 = 2,1,2;
// the mistletoes are xy and yxy, which are already ordered lexicographically
```

```
list L = i1,i2;
ivMis2Base(L); // returns the basis of the factor algebra
↦ [1]:
↦    2
↦ [2]:
↦    2,1
↦ [3]:
↦    2,1,2
↦ [4]:
↦    1
↦ [5]:
↦    1,2
↦ [6]:
↦    0
```

7.7.7.14 ivMis2Dim

Procedure from library `fpadim.lib` (see Section 7.7.7 [fpadim_lib], page 449).

Usage: ivMis2Dim(M); M a list of intvecs

Return: int, the K-dimension of the given algebra

Purpose: Computing the K-dimension out of given mistletoes

Assume: - The mistletoes have to be ordered lexicographically -> OrdMisLex.
Otherwise the returned value may differ from the K-dimension.
- basering is a Letterplace ring.

Example:

```
LIB "fpadim.lib";
ring r = 0,(x,y),dp;
def R = makeLetterplaceRing(5); // constructs a Letterplace ring
R;
↦ // coefficients: QQ
↦ // number of vars : 10
↦ //        block   1 : ordering a
↦ //                  : names    x(1) y(1) x(2) y(2) x(3) y(3) x(4) y(4) x(\
   5) y(5)
↦ //                  : weights     1    1    1    1    1    1    1    1   \
    1    1
↦ //        block   2 : ordering dp
↦ //                  : names    x(1) y(1)
↦ //        block   3 : ordering dp
↦ //                  : names    x(2) y(2)
↦ //        block   4 : ordering dp
↦ //                  : names    x(3) y(3)
↦ //        block   5 : ordering dp
↦ //                  : names    x(4) y(4)
↦ //        block   6 : ordering dp
↦ //                  : names    x(5) y(5)
↦ //        block   7 : ordering C
setring R; // sets basering to Letterplace ring
intvec i1 = 1,2; intvec i2 = 2,1,2;
// the mistletoes are xy and yxy, which are already ordered lexicographically
```

```
list L = i1,i2;
ivMis2Dim(L); // returns the dimension of the factor algebra
↦ 6
```

7.7.7.15 ivOrdMisLex

Procedure from library `fpadim.lib` (see Section 7.7.7 [fpadim_lib], page 449).

Usage: ivOrdMisLex(M); M a list of intvecs

Return: list, containing the ordered intvecs of M

Purpose: Orders a given set of mistletoes lexicographically

Assume: - basering is a Letterplace ring.
- intvecs correspond to monomials

Note: - This is preprocessing, it's not needed if the mistletoes are returned from the sickle algorithm.
- Each entry of the list returned is an intvec.

Example:

```
LIB "fpadim.lib";
ring r = 0,(x,y),dp;
def R = makeLetterplaceRing(5); // constructs a Letterplace ring
setring R; // sets basering to Letterplace ring
intvec i1 = 1,2,1; intvec i2 = 2,2,1; intvec i3 = 1,1; intvec i4 = 2,1,1,1;
// the corresponding monomials are xyx,y^2x,x^2,yx^3
list M = i1,i2,i3,i4;
M;
↦ [1]:
↦    1,2,1
↦ [2]:
↦    2,2,1
↦ [3]:
↦    1,1
↦ [4]:
↦    2,1,1,1
ivOrdMisLex(M);// orders the list of monomials
↦ [1]:
↦    1,1
↦ [2]:
↦    1,2,1
↦ [3]:
↦    2,1,1,1
↦ [4]:
↦    2,2,1
```

7.7.7.16 ivSickle

Procedure from library `fpadim.lib` (see Section 7.7.7 [fpadim_lib], page 449).

Usage: ivSickle(L,n,[degbound]); L a list of intmats, n an int, degbound an optional integer

Return: list, containing intvecs, the mistletoes of A/<L>

Purpose: Computing the mistletoes for a given Groebner basis L

Assume: - basering is a Letterplace ring.
- all rows of each intmat correspond to a Letterplace monomial
- if you specify a different degree bound degbound,
degbound <= attrib(basering,uptodeg) holds.

Note: - If degbound is set, a degree bound will be added. By default there is no degree bound.
- n is the number of variables.
- If the K-dimension is known to be infinite, a degree bound is needed

Example:

```
LIB "fpadim.lib";
ring r = 0,(x,y),dp;
def R = makeLetterplaceRing(5); // constructs a Letterplace ring
setring R; // sets basering to Letterplace ring
//some intmats, which contain monomials in intvec representation as rows
intmat I1 [2][2] = 1,1,2,2; intmat I2 [1][3]  = 1,2,1;
intmat J1 [1][2] =  1,1; intmat J2 [2][3] = 2,1,2,1,2,1;
print(I1);
↦      1     1
↦      2     2
print(I2);
↦      1     2     1
print(J1);
↦      1     1
print(J2);
↦      2     1     2
↦      1     2     1
list G = I1,I2; // ideal, which is already a Groebner basis
list I =  J1,J2; // ideal, which is already a Groebner basis
ivSickle(G,2); // invokes the procedure without any degree bound
↦ [1]:
↦    1,2
↦ [2]:
↦    2,1,2
ivSickle(I,2,5); // invokes the procedure with degree bound 5
↦ [1]:
↦    1,2,2,1
↦ [2]:
↦    1,2,2,2,1
↦ [3]:
↦    1,2,2,2,2
↦ [4]:
↦    2,1
↦ [5]:
↦    2,2,1
↦ [6]:
↦    2,2,2,1
↦ [7]:
↦    2,2,2,2,1
↦ [8]:
↦    2,2,2,2,2
```

7.7.7.17 ivSickleHil

Procedure from library `fpadim.lib` (see Section 7.7.7 [fpadim_lib], page 449).

Usage: ivSickleHil(L,n[,degbound]); L a list of intmats, n an integer, degbound an optional integer

Return: list

Purpose: Computing the mistletoes and the Hilbert series

Assume: - basering is a Letterplace ring.
- all rows of each intmat correspond to a Letterplace monomial
- if you specify a different degree bound degbound, degbound <= attrib(basering,uptodeg) holds.

Note: - If L is the list returned, then L[1] is an intvec, L[2] is a list, containing the mistletoes as intvecs.
- If degbound is set, a degree bound will be added. By default there is no degree bound.
- n is the number of variables.
- If I = L[1] is the intvec returned, then I[k] is the (k-1)-th coefficient of the Hilbert series.
- If the K-dimension is known to be infinite, a degree bound is needed

Example:

```
LIB "fpadim.lib";
ring r = 0,(x,y),dp;
def R = makeLetterplaceRing(5); // constructs a Letterplace ring
setring R; // sets basering to Letterplace ring
//some intmats, which contain monomials in intvec representation as rows
intmat I1[2][2] = 1,1,2,2; intmat I2[1][3]  = 1,2,1;
intmat J1[1][2] =  1,1; intmat J2[2][3] = 2,1,2,1,2,1;
print(I1);
↦      1     1
↦      2     2
print(I2);
↦      1     2     1
print(J1);
↦      1     1
print(J2);
↦      2     1     2
↦      1     2     1
list G = I1,I2;// ideal, which is already a Groebner basis
list I =  J1,J2; // ideal, which is already a Groebner basis
ivSickleHil(G,2); // invokes the procedure without any degree bound
↦ [1]:
↦    1,2,2,1
↦ [2]:
↦    [1]:
↦       1,2
↦    [2]:
↦       2,1,2
ivSickleHil(I,2,5); // invokes the procedure with degree bound 5
↦ [1]:
```

```
↦    1,2,3,3,4,4
↦ [2]:
↦    [1]:
↦       1,2,2,1
↦    [2]:
↦       1,2,2,2,1
↦    [3]:
↦       1,2,2,2,2
↦    [4]:
↦       2,1
↦    [5]:
↦       2,2,1
↦    [6]:
↦       2,2,2,1
↦    [7]:
↦       2,2,2,2,1
↦    [8]:
↦       2,2,2,2,2
```

7.7.7.18 ivSickleDim

Procedure from library `fpadim.lib` (see Section 7.7.7 [fpadim_lib], page 449).

Usage: ivSickleDim(L,n[,degbound]); L a list of intmats, n an integer, degbound an optional integer

Return: list

Purpose: Computing mistletoes and the K-dimension

Assume: - basering is a Letterplace ring.
- all rows of each intmat correspond to a Letterplace monomial
- if you specify a different degree bound degbound,
degbound <= attrib(basering,uptodeg) holds.

Note: - If L is the list returned, then L[1] is an integer, L[2] is a list,
containing the mistletoes as intvecs.
- If degbound is set, a degree bound will be added. By default there
is no degree bound.
- n is the number of variables.
- If the K-dimension is known to be infinite, a degree bound is needed

Example:

```
LIB "fpadim.lib";
ring r = 0,(x,y),dp;
def R = makeLetterplaceRing(5); // constructs a Letterplace ring
setring R; // sets basering to Letterplace ring
//some intmats, which contain monomials in intvec representation as rows
intmat I1 [2][2] = 1,1,2,2; intmat I2 [1][3]  = 1,2,1;
intmat J1 [1][2] =  1,1; intmat J2 [2][3] = 2,1,2,1,2,1;
print(I1);
↦      1     1
↦      2     2
print(I2);
↦      1     2     1
```

```
print(J1);
↦      1     1
print(J2);
↦      2     1     2
↦      1     2     1
list G = I1,I2;// ideal, which is already a Groebner basis
list I =  J1,J2; // ideal, which is already a Groebner basis
ivSickleDim(G,2); // invokes the procedure without any degree bound
↦ [1]:
↦    6
↦ [2]:
↦    [1]:
↦       1,2
↦    [2]:
↦       2,1,2
ivSickleDim(I,2,5); // invokes the procedure with degree bound 5
↦ [1]:
↦    17
↦ [2]:
↦    [1]:
↦       1,2,2,1
↦    [2]:
↦       1,2,2,2,1
↦    [3]:
↦       1,2,2,2,2
↦    [4]:
↦       2,1
↦    [5]:
↦       2,2,1
↦    [6]:
↦       2,2,2,1
↦    [7]:
↦       2,2,2,2,1
↦    [8]:
↦       2,2,2,2,2
```

7.7.7.19 lpOrdMisLex

Procedure from library `fpadim.lib` (see Section 7.7.7 [fpadim_lib], page 449).

Usage: lpOrdMisLex(M); M an ideal of mistletoes

Return: ideal, containing the mistletoes, ordered lexicographically

Purpose: A given set of mistletoes is ordered lexicographically

Assume: - basering is a Letterplace ring.

Note: This is preprocessing, it is not needed if the mistletoes are returned from the sickle algorithm.

Example:

```
LIB "fpadim.lib";
ring r = 0,(x,y),dp;
def R = makeLetterplaceRing(5); // constructs a Letterplace ring
setring R; // sets basering to Letterplace ring
```

```
ideal M = x(1)*y(2)*x(3), y(1)*y(2)*x(3), x(1)*x(2), y(1)*x(2)*x(3)*x(4);
// some monomials
lpOrdMisLex(M); // orders the monomials lexicographically
↦ _[1]=x(1)*x(2)
↦ _[2]=x(1)*y(2)*x(3)
↦ _[3]=y(1)*x(2)*x(3)*x(4)
↦ _[4]=y(1)*y(2)*x(3)
```

7.7.7.20 lpSickle

Procedure from library `fpadim.lib` (see Section 7.7.7 [fpadim_lib], page 449).

Usage: lpSickle(G[,degbound,n]); G an ideal, degbound, n optional integers

Return: ideal

Purpose: Computing the mistletoes of K[X]/<G>

Assume: - basering is a Letterplace ring.
- if you specify a different degree bound degbound,
degbound <= attrib(basering,uptodeg) holds.

Note: - If degbound is set, there will be a degree bound added. 0 means no degree bound. Default: attrib(basering,uptodeg).
- n is the number of variables, which can be set to a different number. Default: attrib(basering, lV).
- If the K-dimension is known to be infinite, a degree bound is needed

Example:

```
LIB "fpadim.lib";
ring r = 0,(x,y),dp;
def R = makeLetterplaceRing(5); // constructs a Letterplace ring
setring R; // sets basering to Letterplace ring
ideal G = x(1)*x(2), y(1)*y(2),x(1)*y(2)*x(3); // ideal G contains a
//Groebner basis
lpSickle(G); //invokes the procedure with ring parameters
↦ _[1]=x(1)*y(2)
↦ _[2]=y(1)*x(2)*y(3)
// the factor algebra is finite, so the degree bound given by the Letterplace
// ring is not necessary
lpSickle(G,0); // procedure without any degree bound
↦ _[1]=x(1)*y(2)
↦ _[2]=y(1)*x(2)*y(3)
```

7.7.7.21 lpSickleHil

Procedure from library `fpadim.lib` (see Section 7.7.7 [fpadim_lib], page 449).

Usage: lpSickleHil(G);

Return: list

Purpose: Computing the Hilbert series and the mistletoes

Assume: - basering is a Letterplace ring.
- if you specify a different degree bound degbound,
degbound <= attrib(basering,uptodeg) holds.

Note: - If L is the list returned, then L[1] is an intvec, corresponding to the Hilbert series, L[2] is an ideal, the mistletoes.
- If degbound is set, there will be a degree bound added. 0 means no degree bound. Default: attrib(basering,uptodeg).
- n is the number of variables, which can be set to a different number. Default: attrib(basering, lV).
- If I = L[1] is the intvec returned, then I[k] is the (k-1)-th coefficient of the Hilbert series.
- If the K-dimension is known to be infinite, a degree bound is needed

Example:

```
LIB "fpadim.lib";
ring r = 0,(x,y),dp;
def R = makeLetterplaceRing(5); // constructs a Letterplace ring
setring R; // sets basering to Letterplace ring
ideal G = x(1)*x(2), y(1)*y(2),x(1)*y(2)*x(3); // ideal G contains a
//Groebner basis
lpSickleHil(G); // invokes the procedure with ring parameters
↦ [1]:
↦    1,2,2,1
↦ [2]:
↦    _[1]=x(1)*y(2)
↦    _[2]=y(1)*x(2)*y(3)
// the factor algebra is finite, so the degree bound given by the Letterplace
// ring is not necessary
lpSickleHil(G,0); // procedure without any degree bound
↦ [1]:
↦    1,2,2,1
↦ [2]:
↦    _[1]=x(1)*y(2)
↦    _[2]=y(1)*x(2)*y(3)
```

7.7.7.22 lpSickleDim

Procedure from library `fpadim.lib` (see Section 7.7.7 [fpadim_lib], page 449).

Usage: lpSickleDim(G[,degbound,n]); G an ideal, degbound, n optional integers

Return: list

Purpose: Computing the K-dimension and the mistletoes

Assume: - basering is a Letterplace ring.
- if you specify a different degree bound degbound, degbound <= attrib(basering,uptodeg) holds.

Note: - If L is the list returned, then L[1] is an integer, the K-dimension, L[2] is an ideal, the mistletoes.
- If degbound is set, there will be a degree bound added. 0 means no degree bound. Default: attrib(basering,uptodeg).
- n is the number of variables, which can be set to a different number. Default: attrib(basering, lV).
- If the K-dimension is known to be infinite, a degree bound is needed

Example:

```
LIB "fpadim.lib";
ring r = 0,(x,y),dp;
def R = makeLetterplaceRing(5); // constructs a Letterplace ring
setring R; // sets basering to Letterplace ring
ideal G = x(1)*x(2), y(1)*y(2),x(1)*y(2)*x(3); // ideal G contains a
//Groebner basis
lpSickleDim(G); // invokes the procedure with ring parameters
↦ [1]:
↦    6
↦ [2]:
↦    _[1]=x(1)*y(2)
↦    _[2]=y(1)*x(2)*y(3)
// the factor algebra is finite, so the degree bound given by the Letterplace
// ring is not necessary
lpSickleDim(G,0); // procedure without any degree bound
↦ [1]:
↦    6
↦ [2]:
↦    _[1]=x(1)*y(2)
↦    _[2]=y(1)*x(2)*y(3)
```

7.7.7.23 sickle

Procedure from library `fpadim.lib` (see Section 7.7.7 [fpadim_lib], page 449).

Usage: sickle(G[,m, d, h, degbound]); G an ideal; m,d,h,degbound optional integers

Return: list

Purpose: Allowing the user to access all procs with one command

Assume: - basering is a Letterplace ring.
- if you specify a different degree bound degbound,
degbound <= attrib(basering,uptodeg) holds.

Note: The returned object will always be a list, but the entries of the returned list may be very different
case m=1,d=1,h=1: see lpDHilbertSickle
case m=1,d=1,h=0: see lpSickleDim
case m=1,d=0,h=1: see lpSickleHil
case m=1,d=0,h=0: see lpSickle (this is the default case)
case m=0,d=1,h=1: see lpDHilbert
case m=0,d=1,h=0: see lpKDim
case m=0,d=0,h=1: see lpHilbert
case m=0,d=0,h=0: returns an error
- If degbound is set, there will be a degree bound added. 0 means no degree bound. Default: attrib(basering,uptodeg).
- If the K-dimension is known to be infinite, a degree bound is needed

Example:

```
LIB "fpadim.lib";
ring r = 0,(x,y),dp;
def R = makeLetterplaceRing(5); // constructs a Letterplace ring
setring R; // sets basering to Letterplace ring
```

```
ideal G = x(1)*x(2), y(1)*y(2),x(1)*y(2)*x(3);
// G contains a Groebner basis
sickle(G,1,1,1); // computes mistletoes, K-dimension and the Hilbert series
↦ [1]:
↦    6
↦ [2]:
↦    1,2,2,1
↦ [3]:
↦    _[1]=x(1)*y(2)
↦    _[2]=y(1)*x(2)*y(3)
sickle(G,1,0,0); // computes mistletoes only
↦ _[1]=x(1)*y(2)
↦ _[2]=y(1)*x(2)*y(3)
sickle(G,0,1,0); // computes K-dimension only
↦ 6
sickle(G,0,0,1); // computes Hilbert series only
↦ 1,2,2,1
```

7.7.7.24 ivMaxIdeal

Procedure from library `fpadim.lib` (see Section 7.7.7 [fpadim_lib], page 449).

Usage: lpMaxIdeal(l, lonly); l an integer, lonly an integer
RETURN: list
PURPOSE: computes a list of free monomials in intvec presentation
with length <= l
if donly `<>` 0, only monomials of degree d are returned ASSUME: - basering is a Letterplace ring.
NOTE: see also lpMaxIdeal()

Example:

```
LIB "fpadim.lib";
ring r = 0,(a,b,c),dp;
def R = makeLetterplaceRing(7); setring R;
ivMaxIdeal(1,0);
↦ [1]:
↦    3
↦ [2]:
↦    2
↦ [3]:
↦    1
ivMaxIdeal(2,0);
↦ [1]:
↦    3
↦ [2]:
↦    2
↦ [3]:
↦    1
↦ [4]:
↦    1,3
↦ [5]:
↦    1,2
↦ [6]:
↦    1,1
```

```
↦ [7]:
↦    2,3
↦ [8]:
↦    2,2
↦ [9]:
↦    2,1
↦ [10]:
↦    3,3
↦ [11]:
↦    3,2
↦ [12]:
↦    3,1
ivMaxIdeal(2,1);
↦ [1]:
↦    1,3
↦ [2]:
↦    1,2
↦ [3]:
↦    1,1
↦ [4]:
↦    2,3
↦ [5]:
↦    2,2
↦ [6]:
↦    2,1
↦ [7]:
↦    3,3
↦ [8]:
↦    3,2
↦ [9]:
↦    3,1
ivMaxIdeal(4,0);
↦ [1]:
↦    3
↦ [2]:
↦    2
↦ [3]:
↦    1
↦ [4]:
↦    1,3
↦ [5]:
↦    1,2
↦ [6]:
↦    1,1
↦ [7]:
↦    2,3
↦ [8]:
↦    2,2
↦ [9]:
↦    2,1
↦ [10]:
↦    3,3
↦ [11]:
```

```
↦    3,2
↦ [12]:
↦    3,1
↦ [13]:
↦    3,1,3
↦ [14]:
↦    3,1,2
↦ [15]:
↦    3,1,1
↦ [16]:
↦    3,2,3
↦ [17]:
↦    3,2,2
↦ [18]:
↦    3,2,1
↦ [19]:
↦    3,3,3
↦ [20]:
↦    3,3,2
↦ [21]:
↦    3,3,1
↦ [22]:
↦    2,1,3
↦ [23]:
↦    2,1,2
↦ [24]:
↦    2,1,1
↦ [25]:
↦    2,2,3
↦ [26]:
↦    2,2,2
↦ [27]:
↦    2,2,1
↦ [28]:
↦    2,3,3
↦ [29]:
↦    2,3,2
↦ [30]:
↦    2,3,1
↦ [31]:
↦    1,1,3
↦ [32]:
↦    1,1,2
↦ [33]:
↦    1,1,1
↦ [34]:
↦    1,2,3
↦ [35]:
↦    1,2,2
↦ [36]:
↦    1,2,1
↦ [37]:
↦    1,3,3
```

```
↦ [38]:
↦    1,3,2
↦ [39]:
↦    1,3,1
↦ [40]:
↦    1,3,1,3
↦ [41]:
↦    1,3,1,2
↦ [42]:
↦    1,3,1,1
↦ [43]:
↦    1,3,2,3
↦ [44]:
↦    1,3,2,2
↦ [45]:
↦    1,3,2,1
↦ [46]:
↦    1,3,3,3
↦ [47]:
↦    1,3,3,2
↦ [48]:
↦    1,3,3,1
↦ [49]:
↦    1,2,1,3
↦ [50]:
↦    1,2,1,2
↦ [51]:
↦    1,2,1,1
↦ [52]:
↦    1,2,2,3
↦ [53]:
↦    1,2,2,2
↦ [54]:
↦    1,2,2,1
↦ [55]:
↦    1,2,3,3
↦ [56]:
↦    1,2,3,2
↦ [57]:
↦    1,2,3,1
↦ [58]:
↦    1,1,1,3
↦ [59]:
↦    1,1,1,2
↦ [60]:
↦    1,1,1,1
↦ [61]:
↦    1,1,2,3
↦ [62]:
↦    1,1,2,2
↦ [63]:
↦    1,1,2,1
↦ [64]:
```

```
↦    1,1,3,3
↦ [65]:
↦    1,1,3,2
↦ [66]:
↦    1,1,3,1
↦ [67]:
↦    2,3,1,3
↦ [68]:
↦    2,3,1,2
↦ [69]:
↦    2,3,1,1
↦ [70]:
↦    2,3,2,3
↦ [71]:
↦    2,3,2,2
↦ [72]:
↦    2,3,2,1
↦ [73]:
↦    2,3,3,3
↦ [74]:
↦    2,3,3,2
↦ [75]:
↦    2,3,3,1
↦ [76]:
↦    2,2,1,3
↦ [77]:
↦    2,2,1,2
↦ [78]:
↦    2,2,1,1
↦ [79]:
↦    2,2,2,3
↦ [80]:
↦    2,2,2,2
↦ [81]:
↦    2,2,2,1
↦ [82]:
↦    2,2,3,3
↦ [83]:
↦    2,2,3,2
↦ [84]:
↦    2,2,3,1
↦ [85]:
↦    2,1,1,3
↦ [86]:
↦    2,1,1,2
↦ [87]:
↦    2,1,1,1
↦ [88]:
↦    2,1,2,3
↦ [89]:
↦    2,1,2,2
↦ [90]:
↦    2,1,2,1
```

```
↦ [91]:
↦    2,1,3,3
↦ [92]:
↦    2,1,3,2
↦ [93]:
↦    2,1,3,1
↦ [94]:
↦    3,3,1,3
↦ [95]:
↦    3,3,1,2
↦ [96]:
↦    3,3,1,1
↦ [97]:
↦    3,3,2,3
↦ [98]:
↦    3,3,2,2
↦ [99]:
↦    3,3,2,1
↦ [100]:
↦    3,3,3,3
↦ [101]:
↦    3,3,3,2
↦ [102]:
↦    3,3,3,1
↦ [103]:
↦    3,2,1,3
↦ [104]:
↦    3,2,1,2
↦ [105]:
↦    3,2,1,1
↦ [106]:
↦    3,2,2,3
↦ [107]:
↦    3,2,2,2
↦ [108]:
↦    3,2,2,1
↦ [109]:
↦    3,2,3,3
↦ [110]:
↦    3,2,3,2
↦ [111]:
↦    3,2,3,1
↦ [112]:
↦    3,1,1,3
↦ [113]:
↦    3,1,1,2
↦ [114]:
↦    3,1,1,1
↦ [115]:
↦    3,1,2,3
↦ [116]:
↦    3,1,2,2
↦ [117]:
```

```
↦    3,1,2,1
↦ [118]:
↦    3,1,3,3
↦ [119]:
↦    3,1,3,2
↦ [120]:
↦    3,1,3,1
ivMaxIdeal(4,1);
↦ [1]:
↦    1,3,1,3
↦ [2]:
↦    1,3,1,2
↦ [3]:
↦    1,3,1,1
↦ [4]:
↦    1,3,2,3
↦ [5]:
↦    1,3,2,2
↦ [6]:
↦    1,3,2,1
↦ [7]:
↦    1,3,3,3
↦ [8]:
↦    1,3,3,2
↦ [9]:
↦    1,3,3,1
↦ [10]:
↦    1,2,1,3
↦ [11]:
↦    1,2,1,2
↦ [12]:
↦    1,2,1,1
↦ [13]:
↦    1,2,2,3
↦ [14]:
↦    1,2,2,2
↦ [15]:
↦    1,2,2,1
↦ [16]:
↦    1,2,3,3
↦ [17]:
↦    1,2,3,2
↦ [18]:
↦    1,2,3,1
↦ [19]:
↦    1,1,1,3
↦ [20]:
↦    1,1,1,2
↦ [21]:
↦    1,1,1,1
↦ [22]:
↦    1,1,2,3
↦ [23]:
```

```
↦    1,1,2,2
↦ [24]:
↦    1,1,2,1
↦ [25]:
↦    1,1,3,3
↦ [26]:
↦    1,1,3,2
↦ [27]:
↦    1,1,3,1
↦ [28]:
↦    2,3,1,3
↦ [29]:
↦    2,3,1,2
↦ [30]:
↦    2,3,1,1
↦ [31]:
↦    2,3,2,3
↦ [32]:
↦    2,3,2,2
↦ [33]:
↦    2,3,2,1
↦ [34]:
↦    2,3,3,3
↦ [35]:
↦    2,3,3,2
↦ [36]:
↦    2,3,3,1
↦ [37]:
↦    2,2,1,3
↦ [38]:
↦    2,2,1,2
↦ [39]:
↦    2,2,1,1
↦ [40]:
↦    2,2,2,3
↦ [41]:
↦    2,2,2,2
↦ [42]:
↦    2,2,2,1
↦ [43]:
↦    2,2,3,3
↦ [44]:
↦    2,2,3,2
↦ [45]:
↦    2,2,3,1
↦ [46]:
↦    2,1,1,3
↦ [47]:
↦    2,1,1,2
↦ [48]:
↦    2,1,1,1
↦ [49]:
↦    2,1,2,3
```

```
↦ [50]:
↦    2,1,2,2
↦ [51]:
↦    2,1,2,1
↦ [52]:
↦    2,1,3,3
↦ [53]:
↦    2,1,3,2
↦ [54]:
↦    2,1,3,1
↦ [55]:
↦    3,3,1,3
↦ [56]:
↦    3,3,1,2
↦ [57]:
↦    3,3,1,1
↦ [58]:
↦    3,3,2,3
↦ [59]:
↦    3,3,2,2
↦ [60]:
↦    3,3,2,1
↦ [61]:
↦    3,3,3,3
↦ [62]:
↦    3,3,3,2
↦ [63]:
↦    3,3,3,1
↦ [64]:
↦    3,2,1,3
↦ [65]:
↦    3,2,1,2
↦ [66]:
↦    3,2,1,1
↦ [67]:
↦    3,2,2,3
↦ [68]:
↦    3,2,2,2
↦ [69]:
↦    3,2,2,1
↦ [70]:
↦    3,2,3,3
↦ [71]:
↦    3,2,3,2
↦ [72]:
↦    3,2,3,1
↦ [73]:
↦    3,1,1,3
↦ [74]:
↦    3,1,1,2
↦ [75]:
↦    3,1,1,1
↦ [76]:
```

```
↦    3,1,2,3
↦ [77]:
↦    3,1,2,2
↦ [78]:
↦    3,1,2,1
↦ [79]:
↦    3,1,3,3
↦ [80]:
↦    3,1,3,2
↦ [81]:
↦    3,1,3,1
```

7.7.7.25 lpMaxIdeal

Procedure from library `fpadim.lib` (see Section 7.7.7 [fpadim_lib], page 449).

Usage: lpMaxIdeal(d, donly); d an integer, donly an integer
RETURN: ideal
PURPOSE: computes a list of free monomials of degree at most d
if donly <> 0, only monomials of degree d are returned ASSUME: - basering is a Letterplace ring.
- d <= attrib(basering,uptodeg) holds. NOTE: analogous to maxideal(d) in the commutative case

Example:

```
LIB "fpadim.lib";
ring r = 0,(a,b,c),dp;
def R = makeLetterplaceRing(7); setring R;
lpMaxIdeal(1,0);
↦ _[1]=c(1)
↦ _[2]=b(1)
↦ _[3]=a(1)
lpMaxIdeal(2,0);
↦ _[1]=c(1)
↦ _[2]=b(1)
↦ _[3]=a(1)
↦ _[4]=a(1)*c(2)
↦ _[5]=a(1)*b(2)
↦ _[6]=a(1)*a(2)
↦ _[7]=b(1)*c(2)
↦ _[8]=b(1)*b(2)
↦ _[9]=b(1)*a(2)
↦ _[10]=c(1)*c(2)
↦ _[11]=c(1)*b(2)
↦ _[12]=c(1)*a(2)
lpMaxIdeal(2,1);
↦ _[1]=a(1)*c(2)
↦ _[2]=a(1)*b(2)
↦ _[3]=a(1)*a(2)
↦ _[4]=b(1)*c(2)
↦ _[5]=b(1)*b(2)
↦ _[6]=b(1)*a(2)
↦ _[7]=c(1)*c(2)
↦ _[8]=c(1)*b(2)
```

```
↦ _[9]=c(1)*a(2)
lpMaxIdeal(4,0);
↦ _[1]=c(1)
↦ _[2]=b(1)
↦ _[3]=a(1)
↦ _[4]=a(1)*c(2)
↦ _[5]=a(1)*b(2)
↦ _[6]=a(1)*a(2)
↦ _[7]=b(1)*c(2)
↦ _[8]=b(1)*b(2)
↦ _[9]=b(1)*a(2)
↦ _[10]=c(1)*c(2)
↦ _[11]=c(1)*b(2)
↦ _[12]=c(1)*a(2)
↦ _[13]=c(1)*a(2)*c(3)
↦ _[14]=c(1)*a(2)*b(3)
↦ _[15]=c(1)*a(2)*a(3)
↦ _[16]=c(1)*b(2)*c(3)
↦ _[17]=c(1)*b(2)*b(3)
↦ _[18]=c(1)*b(2)*a(3)
↦ _[19]=c(1)*c(2)*c(3)
↦ _[20]=c(1)*c(2)*b(3)
↦ _[21]=c(1)*c(2)*a(3)
↦ _[22]=b(1)*a(2)*c(3)
↦ _[23]=b(1)*a(2)*b(3)
↦ _[24]=b(1)*a(2)*a(3)
↦ _[25]=b(1)*b(2)*c(3)
↦ _[26]=b(1)*b(2)*b(3)
↦ _[27]=b(1)*b(2)*a(3)
↦ _[28]=b(1)*c(2)*c(3)
↦ _[29]=b(1)*c(2)*b(3)
↦ _[30]=b(1)*c(2)*a(3)
↦ _[31]=a(1)*a(2)*c(3)
↦ _[32]=a(1)*a(2)*b(3)
↦ _[33]=a(1)*a(2)*a(3)
↦ _[34]=a(1)*b(2)*c(3)
↦ _[35]=a(1)*b(2)*b(3)
↦ _[36]=a(1)*b(2)*a(3)
↦ _[37]=a(1)*c(2)*c(3)
↦ _[38]=a(1)*c(2)*b(3)
↦ _[39]=a(1)*c(2)*a(3)
↦ _[40]=a(1)*c(2)*a(3)*c(4)
↦ _[41]=a(1)*c(2)*a(3)*b(4)
↦ _[42]=a(1)*c(2)*a(3)*a(4)
↦ _[43]=a(1)*c(2)*b(3)*c(4)
↦ _[44]=a(1)*c(2)*b(3)*b(4)
↦ _[45]=a(1)*c(2)*b(3)*a(4)
↦ _[46]=a(1)*c(2)*c(3)*c(4)
↦ _[47]=a(1)*c(2)*c(3)*b(4)
↦ _[48]=a(1)*c(2)*c(3)*a(4)
↦ _[49]=a(1)*b(2)*a(3)*c(4)
↦ _[50]=a(1)*b(2)*a(3)*b(4)
↦ _[51]=a(1)*b(2)*a(3)*a(4)
```

```
↦ _[52]=a(1)*b(2)*b(3)*c(4)
↦ _[53]=a(1)*b(2)*b(3)*b(4)
↦ _[54]=a(1)*b(2)*b(3)*a(4)
↦ _[55]=a(1)*b(2)*c(3)*c(4)
↦ _[56]=a(1)*b(2)*c(3)*b(4)
↦ _[57]=a(1)*b(2)*c(3)*a(4)
↦ _[58]=a(1)*a(2)*a(3)*c(4)
↦ _[59]=a(1)*a(2)*a(3)*b(4)
↦ _[60]=a(1)*a(2)*a(3)*a(4)
↦ _[61]=a(1)*a(2)*b(3)*c(4)
↦ _[62]=a(1)*a(2)*b(3)*b(4)
↦ _[63]=a(1)*a(2)*b(3)*a(4)
↦ _[64]=a(1)*a(2)*c(3)*c(4)
↦ _[65]=a(1)*a(2)*c(3)*b(4)
↦ _[66]=a(1)*a(2)*c(3)*a(4)
↦ _[67]=b(1)*c(2)*a(3)*c(4)
↦ _[68]=b(1)*c(2)*a(3)*b(4)
↦ _[69]=b(1)*c(2)*a(3)*a(4)
↦ _[70]=b(1)*c(2)*b(3)*c(4)
↦ _[71]=b(1)*c(2)*b(3)*b(4)
↦ _[72]=b(1)*c(2)*b(3)*a(4)
↦ _[73]=b(1)*c(2)*c(3)*c(4)
↦ _[74]=b(1)*c(2)*c(3)*b(4)
↦ _[75]=b(1)*c(2)*c(3)*a(4)
↦ _[76]=b(1)*b(2)*a(3)*c(4)
↦ _[77]=b(1)*b(2)*a(3)*b(4)
↦ _[78]=b(1)*b(2)*a(3)*a(4)
↦ _[79]=b(1)*b(2)*b(3)*c(4)
↦ _[80]=b(1)*b(2)*b(3)*b(4)
↦ _[81]=b(1)*b(2)*b(3)*a(4)
↦ _[82]=b(1)*b(2)*c(3)*c(4)
↦ _[83]=b(1)*b(2)*c(3)*b(4)
↦ _[84]=b(1)*b(2)*c(3)*a(4)
↦ _[85]=b(1)*a(2)*a(3)*c(4)
↦ _[86]=b(1)*a(2)*a(3)*b(4)
↦ _[87]=b(1)*a(2)*a(3)*a(4)
↦ _[88]=b(1)*a(2)*b(3)*c(4)
↦ _[89]=b(1)*a(2)*b(3)*b(4)
↦ _[90]=b(1)*a(2)*b(3)*a(4)
↦ _[91]=b(1)*a(2)*c(3)*c(4)
↦ _[92]=b(1)*a(2)*c(3)*b(4)
↦ _[93]=b(1)*a(2)*c(3)*a(4)
↦ _[94]=c(1)*c(2)*a(3)*c(4)
↦ _[95]=c(1)*c(2)*a(3)*b(4)
↦ _[96]=c(1)*c(2)*a(3)*a(4)
↦ _[97]=c(1)*c(2)*b(3)*c(4)
↦ _[98]=c(1)*c(2)*b(3)*b(4)
↦ _[99]=c(1)*c(2)*b(3)*a(4)
↦ _[100]=c(1)*c(2)*c(3)*c(4)
↦ _[101]=c(1)*c(2)*c(3)*b(4)
↦ _[102]=c(1)*c(2)*c(3)*a(4)
↦ _[103]=c(1)*b(2)*a(3)*c(4)
↦ _[104]=c(1)*b(2)*a(3)*b(4)
```

```
↦ _[105]=c(1)*b(2)*a(3)*a(4)
↦ _[106]=c(1)*b(2)*b(3)*c(4)
↦ _[107]=c(1)*b(2)*b(3)*b(4)
↦ _[108]=c(1)*b(2)*b(3)*a(4)
↦ _[109]=c(1)*b(2)*c(3)*c(4)
↦ _[110]=c(1)*b(2)*c(3)*b(4)
↦ _[111]=c(1)*b(2)*c(3)*a(4)
↦ _[112]=c(1)*a(2)*a(3)*c(4)
↦ _[113]=c(1)*a(2)*a(3)*b(4)
↦ _[114]=c(1)*a(2)*a(3)*a(4)
↦ _[115]=c(1)*a(2)*b(3)*c(4)
↦ _[116]=c(1)*a(2)*b(3)*b(4)
↦ _[117]=c(1)*a(2)*b(3)*a(4)
↦ _[118]=c(1)*a(2)*c(3)*c(4)
↦ _[119]=c(1)*a(2)*c(3)*b(4)
↦ _[120]=c(1)*a(2)*c(3)*a(4)
lpMaxIdeal(4,1);
↦ _[1]=a(1)*c(2)*a(3)*c(4)
↦ _[2]=a(1)*c(2)*a(3)*b(4)
↦ _[3]=a(1)*c(2)*a(3)*a(4)
↦ _[4]=a(1)*c(2)*b(3)*c(4)
↦ _[5]=a(1)*c(2)*b(3)*b(4)
↦ _[6]=a(1)*c(2)*b(3)*a(4)
↦ _[7]=a(1)*c(2)*c(3)*c(4)
↦ _[8]=a(1)*c(2)*c(3)*b(4)
↦ _[9]=a(1)*c(2)*c(3)*a(4)
↦ _[10]=a(1)*b(2)*a(3)*c(4)
↦ _[11]=a(1)*b(2)*a(3)*b(4)
↦ _[12]=a(1)*b(2)*a(3)*a(4)
↦ _[13]=a(1)*b(2)*b(3)*c(4)
↦ _[14]=a(1)*b(2)*b(3)*b(4)
↦ _[15]=a(1)*b(2)*b(3)*a(4)
↦ _[16]=a(1)*b(2)*c(3)*c(4)
↦ _[17]=a(1)*b(2)*c(3)*b(4)
↦ _[18]=a(1)*b(2)*c(3)*a(4)
↦ _[19]=a(1)*a(2)*a(3)*c(4)
↦ _[20]=a(1)*a(2)*a(3)*b(4)
↦ _[21]=a(1)*a(2)*a(3)*a(4)
↦ _[22]=a(1)*a(2)*b(3)*c(4)
↦ _[23]=a(1)*a(2)*b(3)*b(4)
↦ _[24]=a(1)*a(2)*b(3)*a(4)
↦ _[25]=a(1)*a(2)*c(3)*c(4)
↦ _[26]=a(1)*a(2)*c(3)*b(4)
↦ _[27]=a(1)*a(2)*c(3)*a(4)
↦ _[28]=b(1)*c(2)*a(3)*c(4)
↦ _[29]=b(1)*c(2)*a(3)*b(4)
↦ _[30]=b(1)*c(2)*a(3)*a(4)
↦ _[31]=b(1)*c(2)*b(3)*c(4)
↦ _[32]=b(1)*c(2)*b(3)*b(4)
↦ _[33]=b(1)*c(2)*b(3)*a(4)
↦ _[34]=b(1)*c(2)*c(3)*c(4)
↦ _[35]=b(1)*c(2)*c(3)*b(4)
↦ _[36]=b(1)*c(2)*c(3)*a(4)
```

```
↦ _[37]=b(1)*b(2)*a(3)*c(4)
↦ _[38]=b(1)*b(2)*a(3)*b(4)
↦ _[39]=b(1)*b(2)*a(3)*a(4)
↦ _[40]=b(1)*b(2)*b(3)*c(4)
↦ _[41]=b(1)*b(2)*b(3)*b(4)
↦ _[42]=b(1)*b(2)*b(3)*a(4)
↦ _[43]=b(1)*b(2)*c(3)*c(4)
↦ _[44]=b(1)*b(2)*c(3)*b(4)
↦ _[45]=b(1)*b(2)*c(3)*a(4)
↦ _[46]=b(1)*a(2)*a(3)*c(4)
↦ _[47]=b(1)*a(2)*a(3)*b(4)
↦ _[48]=b(1)*a(2)*a(3)*a(4)
↦ _[49]=b(1)*a(2)*b(3)*c(4)
↦ _[50]=b(1)*a(2)*b(3)*b(4)
↦ _[51]=b(1)*a(2)*b(3)*a(4)
↦ _[52]=b(1)*a(2)*c(3)*c(4)
↦ _[53]=b(1)*a(2)*c(3)*b(4)
↦ _[54]=b(1)*a(2)*c(3)*a(4)
↦ _[55]=c(1)*c(2)*a(3)*c(4)
↦ _[56]=c(1)*c(2)*a(3)*b(4)
↦ _[57]=c(1)*c(2)*a(3)*a(4)
↦ _[58]=c(1)*c(2)*b(3)*c(4)
↦ _[59]=c(1)*c(2)*b(3)*b(4)
↦ _[60]=c(1)*c(2)*b(3)*a(4)
↦ _[61]=c(1)*c(2)*c(3)*c(4)
↦ _[62]=c(1)*c(2)*c(3)*b(4)
↦ _[63]=c(1)*c(2)*c(3)*a(4)
↦ _[64]=c(1)*b(2)*a(3)*c(4)
↦ _[65]=c(1)*b(2)*a(3)*b(4)
↦ _[66]=c(1)*b(2)*a(3)*a(4)
↦ _[67]=c(1)*b(2)*b(3)*c(4)
↦ _[68]=c(1)*b(2)*b(3)*b(4)
↦ _[69]=c(1)*b(2)*b(3)*a(4)
↦ _[70]=c(1)*b(2)*c(3)*c(4)
↦ _[71]=c(1)*b(2)*c(3)*b(4)
↦ _[72]=c(1)*b(2)*c(3)*a(4)
↦ _[73]=c(1)*a(2)*a(3)*c(4)
↦ _[74]=c(1)*a(2)*a(3)*b(4)
↦ _[75]=c(1)*a(2)*a(3)*a(4)
↦ _[76]=c(1)*a(2)*b(3)*c(4)
↦ _[77]=c(1)*a(2)*b(3)*b(4)
↦ _[78]=c(1)*a(2)*b(3)*a(4)
↦ _[79]=c(1)*a(2)*c(3)*c(4)
↦ _[80]=c(1)*a(2)*c(3)*b(4)
↦ _[81]=c(1)*a(2)*c(3)*a(4)
```

7.7.7.26 monomialBasis

Procedure from library `fpadim.lib` (see Section 7.7.7 [fpadim_lib], page 449).

Usage: monomialBasis(d, donly, J); d, donly integers, J an ideal RETURN: ideal
PURPOSE: computes a list of free monomials in a Letterplace
basering R of degree at most d and not contained in <LM(J)>
if donly <> 0, only monomials of degree d are returned ASSUME: - basering is a

Letterplace ring.
- d <= attrib(basering,uptodeg) holds.
- J is a Groebner basis

Example:

```
LIB "fpadim.lib";
ring r = 0,(x,y),dp;
def R = makeLetterplaceRing(7); setring R;
ideal J = x(1)*y(2)*x(3) - y(1)*x(2)*y(3);
option(redSB); option(redTail);
J = letplaceGBasis(J);
J;
↦ J[1]=x(1)*y(2)*x(3)-y(1)*x(2)*y(3)
↦ J[2]=x(1)*y(2)*y(3)*x(4)*y(5)-y(1)*x(2)*y(3)*y(4)*x(5)
↦ J[3]=x(1)*y(2)*y(3)*y(4)*x(5)*y(6)-y(1)*x(2)*y(3)*y(4)*x(5)*x(6)
↦ J[4]=x(1)*y(2)*y(3)*y(4)*y(5)*x(6)*y(7)-y(1)*x(2)*y(3)*y(4)*x(5)*x(6)*x(7\
   )
monomialBasis(2,1,std(0));
↦ _[1]=x(1)*y(2)
↦ _[2]=x(1)*x(2)
↦ _[3]=y(1)*y(2)
↦ _[4]=y(1)*x(2)
↦ // ** right side is not a datum, assignment ignored
↦ // ** in line >>;return();<<
↦ _[1]=0
monomialBasis(2,0,std(0));
↦ _[1]=y(1)
↦ _[2]=x(1)
↦ _[3]=x(1)*y(2)
↦ _[4]=x(1)*x(2)
↦ _[5]=y(1)*y(2)
↦ _[6]=y(1)*x(2)
↦ // ** right side is not a datum, assignment ignored
↦ // ** in line >>;return();<<
↦ _[1]=1
↦ _[2]=0
monomialBasis(3,1,J);
↦ _[1]=y(1)*y(2)*y(3)
↦ _[2]=y(1)*y(2)*x(3)
↦ _[3]=x(1)*y(2)*y(3)
↦ _[4]=y(1)*x(2)*y(3)
↦ _[5]=y(1)*x(2)*x(3)
↦ _[6]=x(1)*x(2)*y(3)
↦ _[7]=x(1)*x(2)*x(3)
monomialBasis(3,0,J);
↦ _[1]=1
↦ _[2]=y(1)
↦ _[3]=y(1)
↦ _[4]=x(1)
↦ _[5]=x(1)
↦ _[6]=y(1)*y(2)
↦ _[7]=y(1)*x(2)
↦ _[8]=x(1)*y(2)
```

```
↦ _[9]=x(1)*x(2)
↦ _[10]=y(1)*y(2)*y(3)
↦ _[11]=y(1)*y(2)*x(3)
↦ _[12]=y(1)*x(2)*y(3)
↦ _[13]=y(1)*x(2)*x(3)
↦ _[14]=x(1)*y(2)*y(3)
↦ _[15]=x(1)*x(2)*y(3)
↦ _[16]=x(1)*x(2)*x(3)
```

7.7.7.27 ivL2lpI

Procedure from library `fpadim.lib` (see Section 7.7.7 [fpadim_lib], page 449).

Usage: ivL2lpI(L); L a list of intvecs

Return: ideal

Purpose: Transforming a list of intvecs into an ideal of Letterplace monomials

Assume: - Intvec corresponds to a Letterplace monomial
- basering has to be a Letterplace ring

Note: - Assumptions will not be checked!

Example:

```
LIB "fpadim.lib";
ring r = 0,(x,y,z),dp;
def R = makeLetterplaceRing(5);// constructs a Letterplace ring
setring R; //sets basering to Letterplace ring
intvec u = 1,1,2; intvec v = 2,1,3; intvec w = 3,1,1;
// u = x^2y, v = yxz, w = zx^2 in intvec representation
list L = u,v,w;
ivL2lpI(L);// invokes the procedure, returns the ideal containing u,v,w
↦ _[1]=x(1)*x(2)*y(3)
↦ _[2]=y(1)*x(2)*z(3)
↦ _[3]=z(1)*x(2)*x(3)
```

7.7.7.28 iv2lp

Procedure from library `fpadim.lib` (see Section 7.7.7 [fpadim_lib], page 449).

Usage: iv2lp(I); I an intvec

Return: poly

Purpose: Transforming an intvec into the corresponding Letterplace polynomial

Assume: - Intvec corresponds to a Letterplace monomial
- basering has to be a Letterplace ring

Note: - Assumptions will not be checked!

Example:

```
LIB "fpadim.lib";
ring r = 0,(x,y,z),dp;
def R = makeLetterplaceRing(5); // constructs a Letterplace ring
setring R; //sets basering to Letterplace ring
intvec u = 1,1,2; intvec v = 2,1,3; intvec w = 3,1,1;
```

```
// u = x^2y, v = yxz, w = zx^2 in intvec representation
iv2lp(u); // invokes the procedure and returns the corresponding poly
↦ x(1)*x(2)*y(3)
iv2lp(v);
↦ y(1)*x(2)*z(3)
iv2lp(w);
↦ z(1)*x(2)*x(3)
```

7.7.7.29 iv2lpList

Procedure from library `fpadim.lib` (see Section 7.7.7 [fpadim_lib], page 449).

Usage: iv2lpList(L); L a list of intmats

Return: ideal

Purpose: Converting a list of intmats into an ideal of corresponding monomials

Assume: - The rows of each intmat in L must correspond to a Letterplace monomial
- basering has to be a Letterplace ring

Example:

```
LIB "fpadim.lib";
ring r = 0,(x,y,z),dp;
def R = makeLetterplaceRing(5); // constructs a Letterplace ring
setring R; // sets basering to Letterplace ring
intmat u[3][1] = 1,1,2; intmat v[1][3] = 2,1,3; intmat w[2][3] = 3,1,1,2,3,1;
// defines intmats of different size containing intvec representations of
// monomials as rows
list L = u,v,w;
print(u); print(v); print(w); // shows the intmats contained in L
↦      1
↦      1
↦      2
↦      2     1     3
↦      3     1     1
↦      2     3     1
iv2lpList(L); // returns the corresponding monomials as an ideal
↦ _[1]=x(1)
↦ _[2]=y(1)
↦ _[3]=y(1)*x(2)*z(3)
↦ _[4]=z(1)*x(2)*x(3)
↦ _[5]=y(1)*z(2)*x(3)
```

7.7.7.30 iv2lpMat

Procedure from library `fpadim.lib` (see Section 7.7.7 [fpadim_lib], page 449).

Usage: iv2lpMat(M); M an intmat

Return: ideal

Purpose: Converting an intmat into an ideal of the corresponding monomials

Assume: - The rows of M must correspond to Letterplace monomials
- basering has to be a Letterplace ring

Example:

```
LIB "fpadim.lib";
ring r = 0,(x,y,z),dp;
def R = makeLetterplaceRing(5); // constructs a Letterplace ring
setring R; // sets basering to Letterplace ring
intmat u[3][1] = 1,1,2; intmat v[1][3] = 2,1,3; intmat w[2][3] = 3,1,1,2,3,1;
// defines intmats of different size containing intvec representations of
// monomials as rows
iv2lpMat(u); // returns the monomials contained in u
↦ _[1]=x(1)
↦ _[2]=x(1)
↦ _[3]=y(1)
iv2lpMat(v); // returns the monomials contained in v
↦ _[1]=y(1)*x(2)*z(3)
iv2lpMat(w); // returns the monomials contained in w
↦ _[1]=z(1)*x(2)*x(3)
↦ _[2]=y(1)*z(2)*x(3)
```

7.7.7.31 lp2iv

Procedure from library `fpadim.lib` (see Section 7.7.7 [fpadim_lib], page 449).

Usage: lp2iv(p); p a poly

Return: intvec

Purpose: Transforming a monomial into the corresponding intvec

Assume: - basering has to be a Letterplace ring

Note: - Assumptions will not be checked!

Example:

```
LIB "fpadim.lib";
ring r = 0,(x,y,z),dp;
def R = makeLetterplaceRing(5); // constructs a Letterplace ring
setring R; // sets basering to Letterplace ring
poly p = x(1)*x(2)*z(3); poly q = y(1)*y(2)*x(3)*x(4);
poly w= z(1)*y(2)*x(3)*z(4)*z(5);
// p,q,w are some polynomials we want to transform into their
// intvec representation
lp2iv(p); lp2iv(q); lp2iv(w);
↦ 1,1,3
↦ 2,2,1,1
↦ 3,2,1,3,3
```

7.7.7.32 lp2ivId

Procedure from library `fpadim.lib` (see Section 7.7.7 [fpadim_lib], page 449).

Usage: lp2ivId(G); G an ideal

Return: list

Purpose: Converting an ideal into an list of intmats,
the corresponding intvecs forming the rows

Assume: - basering has to be a Letterplace ring

Example:

```
LIB "fpadim.lib";
ring r = 0,(x,y,z),dp;
def R = makeLetterplaceRing(5); // constructs a Letterplace ring
setring R; // sets basering to Letterplace ring
poly p = x(1)*x(2)*z(3); poly q = y(1)*y(2)*x(3)*x(4);
poly w = z(1)*y(2)*x(3)*z(4);
// p,q,w are some polynomials we want to transform into their
// intvec representation
ideal G = p,q,w;
// define the ideal containing p,q and w
lp2ivId(G); // and return the list of intmats for this ideal
↦ [1]:
↦    1,1,3
↦ [2]:
↦    2,2,1,1,
↦    3,2,1,3
```

7.7.7.33 lpId2ivLi

Procedure from library `fpadim.lib` (see Section 7.7.7 [fpadim_lib], page 449).

Usage: lpId2ivLi(G); G an ideal

Return: list

Purpose: Transforming an ideal into the corresponding list of intvecs

Assume: - basering has to be a Letterplace ring

Example:

```
LIB "fpadim.lib";
ring r = 0,(x,y),dp;
def R = makeLetterplaceRing(5); // constructs a Letterplace ring
setring R; // sets basering to Letterplace ring
ideal L = x(1)*x(2),y(1)*y(2),x(1)*y(2)*x(3);
lpId2ivLi(L); // returns the corresponding intvecs as a list
↦ [1]:
↦    1,1
↦ [2]:
↦    2,2
↦ [3]:
↦    1,2,1
```

7.7.8 fpaprops_lib

Library: fpaprops.lib

Purpose: Algorithms for the properties of quotient algebras in the letterplace case

Authors: Karim Abou Zeid, karim.abou.zeid at rwth-aachen.de

Support: Project II.6 in the transregional collaborative research centre SFB-TRR 195 'Symbolic Tools in Mathematics and their Application' of the German DFG

Overview: Algorithms for computing various properties of quotient algebras in the letterplace case.

References:
Huishi Li: Groebner bases in ring theory. World Scientific, 2010.

Procedures: See also: Section 7.7.7 [fpadim_lib], page 449; Section 7.7.9 [freegb_lib], page 495.

7.7.8.1 lpNoetherian

Procedure from library `fpaprops.lib` (see Section 7.7.8 [fpaprops_lib], page 490).

Usage: lpNoetherian(G); G an ideal in a Letterplace ring

Return: int
0 not noetherian
1 left noetherian
2 right noetherian
3 noetherian

Purpose: Check whether R/<G> is (left/right) noetherian, where R is the basering

Assume: - basering is a Letterplace ring
- G is a Groebner basis

Example:

```
LIB "fpaprops.lib";
ring r = 0,(x,y),dp;
def R = makeLetterplaceRing(5);
setring R;
ideal G = x(1)*x(2), y(1)*x(2); // K<x,y>/<xx,yx> is right noetherian
lpNoetherian(G);
↦ 2
```

7.7.8.2 lpIsSemiPrime

Procedure from library `fpaprops.lib` (see Section 7.7.8 [fpaprops_lib], page 490).

Usage: lpIsSemiPrime(G); G an ideal in a Letterplace ring

Return: boolean

Purpose: Check whether R/<G> is semi prime, where R is the basering

Assume: - basering is a Letterplace ring
- G is a Groebner basis

Example:

```
LIB "fpaprops.lib";
ring r = 0,(x1,x2),dp;
def R = makeLetterplaceRing(5);
setring R;
ideal G = x1(1)*x2(2), x2(1)*x1(2); // K<x1,x2>/<x1*x2,x2*x1> is semi prime
lpIsSemiPrime(G);
↦ 1
```

7.7.8.3 lpIsPrime

Procedure from library `fpaprops.lib` (see Section 7.7.8 [fpaprops_lib], page 490).

Usage: lpIsPrime(G); G an ideal in a Letterplace ring

Return: boolean

Purpose: Check whether R/<G> is prime, where R is the basering

Assume: - basering is a Letterplace ring
- G is a Groebner basis

Example:

```
LIB "fpaprops.lib";
ring r = 0,(x,y),dp;
def R = makeLetterplaceRing(5);
setring R;
ideal G = x(1)*x(2), y(1)*y(2); // K<x,y>/<xx,yy> is prime
lpIsPrime(G);
↦ 1
```

7.7.8.4 lpGkDim

Procedure from library `fpaprops.lib` (see Section 7.7.8 [fpaprops_lib], page 490).

Usage: lpGkDim(G); G an ideal in a letterplace ring

Return: int

Purpose: Determines the Gelfand Kirillov dimension of A/<G>
-1 means it is positive infinite

Assume: - basering is a Letterplace ring
- G is a Groebner basis

Example:

```
LIB "fpaprops.lib";
ring r = 0,(x,y,z),dp;
def R = makeLetterplaceRing(5); // constructs a Letterplace ring
R;
↦ // coefficients: QQ
↦ // number of vars : 15
↦ //        block   1 : ordering a
↦ //                  : names    x(1) y(1) z(1) x(2) y(2) z(2) x(3) y(3) z(\
   3) x(4) y(4) z(4) x(5) y(5) z(5)
↦ //                  : weights     1    1    1    1    1    1    1    1   \
    1    1    1    1    1    1    1
↦ //        block   2 : ordering dp
↦ //                  : names    x(1) y(1) z(1)
↦ //        block   3 : ordering dp
↦ //                  : names    x(2) y(2) z(2)
↦ //        block   4 : ordering dp
↦ //                  : names    x(3) y(3) z(3)
↦ //        block   5 : ordering dp
↦ //                  : names    x(4) y(4) z(4)
↦ //        block   6 : ordering dp
```

```
↦ //                  : names    x(5) y(5) z(5)
↦ //        block   7 : ordering C
setring R; // sets basering to Letterplace ring
ideal I = z(1);//an example of infinite GK dimension
lpGkDim(I);
↦ -1
I = x(1),y(1),z(1); // gkDim = 0
lpGkDim(I);
↦ 0
I = x(1)*y(2), x(1)*z(2), z(1)*y(2), z(1)*z(2);//gkDim = 2
lpGkDim(I);
↦ 2
```

7.7.8.5 lpGlDimBound

Procedure from library `fpaprops.lib` (see Section 7.7.8 [fpaprops_lib], page 490).

Usage: lpGlDimBound(I); I an ideal

Return: int, an upper bound for the global dimension, -1 means infinity

Purpose: computing an upper bound for the global dimension

Assume: - basering is a Letterplace ring, G is a reduced Groebner Basis

Note: if I = LM(I), then the global dimension is equal the Gelfand Kirillov dimension if it is finite
Global dimension should be 0 for A/G = K and 1 for A/G = K<x1...xn>

Example:

```
LIB "fpaprops.lib";
ring r = 0,(x,y),dp;
def R = makeLetterplaceRing(5); // constructs a Letterplace ring
setring R; // sets basering to Letterplace ring
ideal G = x(1)*x(2), y(1)*y(2),x(1)*y(2)*x(3); // ideal G contains a
//Groebner basis
lpGlDimBound(G); // invokes procedure with Groebner basis G
↦ 0
```

7.7.8.6 lpSubstitute

Procedure from library `fpaprops.lib` (see Section 7.7.8 [fpaprops_lib], page 490).

Usage: lpSubstitute(f,s1,s2[,G]); f letterplace polynomial, s1 list (ideal) of variables to replace, s2 list (ideal) of polynomials to replace with, G optional ideal to reduce with.

Return: poly, the substituted polynomial

Assume: - basering is a Letterplace ring
- G is a groebner basis,
- the current ring has a sufficient degbound (can be calculated with lpCalcSubstDegBound())

Example:

```
LIB "fpaprops.lib";
//////// EXAMPLE A ////////
ring r = 0,(x,y,z),dp;
```

```
def R = makeLetterplaceRing(4);
setring R;
ideal G = x(1)*y(2); // optional
poly f = 3*x(1)*x(2)+y(1)*x(2);
ideal s1 = x(1), y(1);
ideal s2 = y(1)*z(2)*z(3), x(1);
// the substitution probably needs a higher degbound
int minDegBound = lpCalcSubstDegBounds(f,s1,s2);
↦ // ** int division with ‘/‘: use ‘div‘ instead in line >>  maxDegBound = \
   maxDegBound + maxDegBound/2;<<
setring r;
def R1 = makeLetterplaceRing(minDegBound);
setring R1;
// the last parameter is optional
lpSubstitute(imap(R,f), imap(R,s1), imap(R,s2), imap(R,G));
↦ y(1)*z(2)*z(3)*y(4)*z(5)*z(6)
//////// EXAMPLE B ////////
ring r = 0,(x,y,z),dp;
↦ // ** redefining r (ring r = 0,(x,y,z),dp;)
def R = makeLetterplaceRing(4);
↦ // ** redefining R (def R = makeLetterplaceRing(4);)
setring R;
poly f = 3*x(1)*x(2)+y(1)*x(2);
poly g = z(1)*x(2)+y(1);
poly h = 7*x(1)*z(2)+x(1);
ideal I = f,g,h;
ideal s1 = x(1), y(1);
ideal s2 = y(1)*z(2)*z(3), x(1);
int minDegBound = lpCalcSubstDegBounds(I,s1,s2);
↦ // ** redefining minDegBound (int minDegBound = lpCalcSubstDegBounds(I,s1\
   ,s2);)
↦ // ** int division with ‘/‘: use ‘div‘ instead in line >>  maxDegBound = \
   maxDegBound + maxDegBound/2;<<
↦ // ** int division with ‘/‘: use ‘div‘ instead in line >>  maxDegBound = \
   maxDegBound + maxDegBound/2;<<
↦ // ** int division with ‘/‘: use ‘div‘ instead in line >>  maxDegBound = \
   maxDegBound + maxDegBound/2;<<
setring r;
def R1 = makeLetterplaceRing(minDegBound);
↦ // ** redefining R1 (def R1 = makeLetterplaceRing(minDegBound);)
setring R1;
ideal I = imap(R,I);
ideal s1 = imap(R,s1);
ideal s2 = imap(R,s2);
for (int i = 1; i <= size(I); i++) {
lpSubstitute(I[i], s1, s2);
}
↦ 3*y(1)*z(2)*z(3)*y(4)*z(5)*z(6)+x(1)*y(2)*z(3)*z(4)
↦ z(1)*y(2)*z(3)*z(4)+x(1)
↦ 7*y(1)*z(2)*z(3)*z(4)+y(1)*z(2)*z(3)
```

7.7.8.7 lpCalcSubstDegBound

Procedure from library `fpaprops.lib` (see Section 7.7.8 [fpaprops_lib], page 490).

Usage: lpCalcSubstDegBound(f,s1,s2); f letterplace polynomial, s1 list (ideal) of variables to replace, s2 list (ideal) of polynomials to replace with

Return: int, the min degbound required to perform the substitution

Assume: - basering is a Letterplace ring

Example:

```
LIB "fpaprops.lib";
// see lpSubstitute()
```

7.7.8.8 lpCalcSubstDegBounds

Procedure from library `fpaprops.lib` (see Section 7.7.8 [fpaprops_lib], page 490).

Usage: lpCalcSubstDegBounds(I,s1,s2); I list (ideal) of letterplace polynomials, s1 list (ideal) of variables to replace, s2 list (ideal) of polynomials to replace with

Return: int, the min degbound required to perform all of the substitutions

Assume: - basering is a Letterplace ring

Note: convenience method

Example:

```
LIB "fpaprops.lib";
// see lpSubstitute()
```

7.7.9 freegb_lib

Library: freegb.lib

Purpose: Compute two-sided Groebner bases in free algebras via letterplace approach

Authors: Viktor Levandovskyy, viktor.levandovskyy@math.rwth-aachen.de Grischa Studzinski, grischa.studzinski@math.rwth-aachen.de

Overview: For the theory, see chapter 'Letterplace' in the @sc{Singular} Manual

Support: Joint projects LE 2697/2-1 and KR 1907/3-1 of the Priority Programme SPP 1489: 'Algorithmische und Experimentelle Methoden in Algebra, Geometrie und Zahlentheorie' of the German DFG
and Project II.6 of the transregional collaborative research centre SFB-TRR 195 'Symbolic Tools in Mathematics and their Application' of the German DFG

Procedures: See also: Section 7.6 [LETTERPLACE], page 367; Section 7.7.7 [fpadim_lib], page 449.

7.7.9.1 makeLetterplaceRing

Procedure from library `freegb.lib` (see Section 7.7.9 [freegb_lib], page 495).

Usage: makeLetterplaceRing(d [,h]); d an integer, h an optional integer

Return: ring

Purpose: creates a ring with the ordering, used in letterplace computations

Note: h = 0 (default) : Dp ordering will be used
h = 2 : weights 1 used for all the variables, a tie breaker is a list of block of original ring
h = 1 : the pure homogeneous letterplace block ordering (applicable in the situation of homogeneous input ideals) will be used.

Example:

```
LIB "freegb.lib";
ring r = 0,(x,y,z),(dp(1),dp(2));
def A = makeLetterplaceRing(2);
setring A;  A;
↦ // coefficients: QQ
↦ // number of vars : 6
↦ //        block   1 : ordering a
↦ //                  : names    x(1) y(1) z(1) x(2) y(2) z(2)
↦ //                  : weights     1    1    1    1    1    1
↦ //        block   2 : ordering lp
↦ //                  : names    x(1)
↦ //        block   3 : ordering dp
↦ //                  : names    y(1) z(1)
↦ //        block   4 : ordering lp
↦ //                  : names    x(2)
↦ //        block   5 : ordering dp
↦ //                  : names    y(2) z(2)
↦ //        block   6 : ordering C
attrib(A,"isLetterplaceRing");
↦ 1
attrib(A,"uptodeg");  // degree bound
↦ 2
attrib(A,"lV"); // number of variables in the main block
↦ 3
setring r; def B = makeLetterplaceRing(2,1); // to compare:
setring B;  B;
↦ // coefficients: QQ
↦ // number of vars : 6
↦ //        block   1 : ordering lp
↦ //                  : names    x(1)
↦ //        block   2 : ordering dp
↦ //                  : names    y(1) z(1)
↦ //        block   3 : ordering lp
↦ //                  : names    x(2)
↦ //        block   4 : ordering dp
↦ //                  : names    y(2) z(2)
↦ //        block   5 : ordering C
```

7.7.9.2 letplaceGBasis

Procedure from library `freegb.lib` (see Section 7.7.9 [freegb_lib], page 495).

Usage: letplaceGBasis(I); I an ideal

Return: ideal

Assume: basering is a Letterplace ring, an ideal consists of Letterplace polynomials

Purpose: compute the two-sided Groebner basis of an ideal I via Letterplace algorithm

Note: the degree bound for this computation is read off the letterplace structure of basering

Example:

```
LIB "freegb.lib";
ring r = 0,(x,y,z),(dp(1),dp(2));
int degree_bound = 5;
def R = makeLetterplaceRing(5);
setring R;
ideal I = -x(1)*y(2)-7*y(1)*y(2)+3*x(1)*x(2), x(1)*y(2)*x(3)-y(1)*x(2)*y(3);
ideal J = letplaceGBasis(I);
J;
↦ J[1]=3*x(1)*x(2)-x(1)*y(2)-7*y(1)*y(2)
↦ J[2]=22*x(1)*y(2)*y(3)-3*y(1)*x(2)*y(3)-21*y(1)*y(2)*x(3)+7*y(1)*y(2)*y(3\
   )
↦ J[3]=3*x(1)*y(2)*x(3)-22*x(1)*y(2)*y(3)+21*y(1)*y(2)*x(3)-7*y(1)*y(2)*y(3\
   )
↦ J[4]=22803*y(1)*y(2)*y(3)*x(4)+19307*y(1)*y(2)*y(3)*y(4)
↦ J[5]=1933*y(1)*y(2)*x(3)*y(4)+2751*y(1)*y(2)*y(3)*x(4)+161*y(1)*y(2)*y(3)\
   *y(4)
↦ J[6]=y(1)*y(2)*y(3)*y(4)*y(5)
// now transfom letterplace polynomials into strings of words
lp2lstr(J,r); // export an object called @code{@LN} to the ring r
setring r;  // change to the ring r
lst2str(@LN,1);
↦ [1]:
↦    3*x*x-x*y-7*y*y
↦ [2]:
↦    22*x*y*y-3*y*x*y-21*y*y*x+7*y*y*y
↦ [3]:
↦    3*x*y*x-22*x*y*y+21*y*y*x-7*y*y*y
↦ [4]:
↦    22803*y*y*y*x+19307*y*y*y*y
↦ [5]:
↦    1933*y*y*x*y+2751*y*y*y*x+161*y*y*y*y
↦ [6]:
↦    y*y*y*y*y
```

7.7.9.3 lpNF

Procedure from library `freegb.lib` (see Section 7.7.9 [freegb_lib], page 495).

Usage: lpNF(p,G); f letterplace polynomial, ideal I

Return: poly

Purpose: computation of the normal form of p with respect to G

Assume: p is a Letterplace polynomial, G is a set Letterplace polynomials, being a Letterplace Groebner basis (no check for this will be done)

Note: Strategy: take the smallest monomial wrt ordering for reduction - For homogenous ideals the shift does not matter
- For non-homogenous ideals the first shift will be the smallest monomial

Example:

```
LIB "freegb.lib";
ring r = 0,(x,y,z),dp;
int d =5; // degree
def R = makeLetterplaceRing(d);
setring R;
ideal I = y(1)*x(2)*y(3) - z(1)*y(2)*z(3), x(1)*y(2)*x(3) - z(1)*x(2)*y(3), z(1)*x(2)*z(3) - y(1)*z(2)*x(3), x(1)*x(2)*x(3) + y(1)*y(2)*y(3) + z(1)*z(2)*z(3) + x(1)*y(2)*z(3);
ideal J = letplaceGBasis(I); // compute a Letterplace Groebner basis
poly p = y(1)*x(2)*y(3)*z(4)*y(5) - y(1)*z(2)*z(3)*y(4) + z(1)*y(2)*z(3);
poly q = z(1)*x(2)*z(3)*y(4)*z(5) - y(1)*z(2)*x(3)*y(4)*z(5);
lpNF(p,J);
↦ z(1)*y(2)*z(3)*z(4)*y(5)-y(1)*z(2)*z(3)*y(4)+z(1)*y(2)*z(3)
lpNF(q,J);
↦ 0
```

7.7.9.4 setLetterplaceAttributes

Procedure from library `freegb.lib` (see Section 7.7.9 [freegb_lib], page 495).

Usage: setLetterplaceAttributes(R, d, b); R a ring, b,d integers

Return: ring with special attributes set

Purpose: sets attributes for a letterplace ring:
'isLetterplaceRing' = true, 'uptodeg' = d, 'lV' = b, where
'uptodeg' stands for the degree bound,
'lV' for the number of variables in the block 0.

Note: Activate the resulting ring by using `setring`

Example:

```
LIB "freegb.lib";
ring r = 0,(x(1),y(1),x(2),y(2),x(3),y(3),x(4),y(4)),dp;
def R = setLetterplaceAttributes(r, 4, 2); setring R;
attrib(R,"isLetterplaceRing");
↦ 1
lieBracket(x(1),y(1),2);
↦ -y(1)*x(2)*x(3)*x(4)+3*x(1)*y(2)*x(3)*x(4)-3*x(1)*x(2)*y(3)*x(4)+x(1)*x(2\
   )*x(3)*y(4)
```

7.7.9.5 freeGBasis

Procedure from library `freegb.lib` (see Section 7.7.9 [freegb_lib], page 495).

Usage: freeGBasis(L, d); L a list of modules, d an integer

Return: ring

Assume: L has a special form. Namely, it is a list of modules, where

- each generator of every module stands for a monomial times coefficient in free algebra,

- in such a vector generator, the 1st entry is a nonzero coefficient from the ground field

- and each next entry hosts a variable from the basering.

Purpose: compute the two-sided Groebner basis of an ideal, encoded by L in the free associative algebra, up to degree d

Note: Apply `lst2str` to the output in order to obtain a better readable presentation

Example:

```
LIB "freegb.lib";
ring r = 0,(x,y,z),(dp(1),dp(2)); //  ring r = 0,(x,y,z),(a(3,0,2), dp(2));
module M = [-1,x,y],[-7,y,y],[3,x,x]; // stands for free poly -xy - 7yy - 3xx
module N = [1,x,y,x],[-1,y,x,y]; // stands for free poly xyx - yxy
list L; L[1] = M; L[2] = N; // list of modules stands for an ideal in free algebra
lst2str(L); // list to string conversion of input polynomials
↦ [1]:
↦    -xy-7yy+3xx
↦ [2]:
↦    xyx-yxy
def U = freeGBasis(L,5); // 5 is the degree bound
lst2str(U);
↦ [1]:
↦    yyyyy
↦ [2]:
↦    22803yyyx+19307yyyy
↦ [3]:
↦    1933yyxy+2751yyyx+161yyyy
↦ [4]:
↦    22xyy-3yxy-21yyx+7yyy
↦ [5]:
↦    3xyx-22xyy+21yyx-7yyy
↦ [6]:
↦    3xx-xy-7yy
```

7.7.9.6 lpMult

Procedure from library `freegb.lib` (see Section 7.7.9 [freegb_lib], page 495).

Usage: lpMult(f,g); f,g letterplace polynomials

Return: poly

Assume: basering has a letterplace ring structure

Purpose: compute the letterplace form of f*g

Example:

```
LIB "freegb.lib";
// define a ring in letterplace form as follows:
ring r = 0,(x(1),y(1),x(2),y(2),x(3),y(3),x(4),y(4)),dp;
def R = setLetterplaceAttributes(r,4,2); // supply R with letterplace structure
setring R;
poly a = x(1)*y(2)+x(1)+y(1); poly b = y(1)+3;
lpMult(b,a);
↦ y(1)*x(2)*y(3)+y(1)*x(2)+3*x(1)*y(2)+y(1)*y(2)+3*x(1)+3*y(1)
lpMult(a,b);
↦ x(1)*y(2)*y(3)+4*x(1)*y(2)+y(1)*y(2)+3*x(1)+3*y(1)
```

7.7.9.7 shiftPoly

Procedure from library **freegb.lib** (see Section 7.7.9 [freegb_lib], page 495).

Usage: shiftPoly(p,i); p letterplace poly, i int

Return: poly

Assume: basering has letterplace ring structure

Purpose: compute the i-th shift of letterplace polynomial p

Example:

```
LIB "freegb.lib";
ring r = 0,(x,y,z),dp;
int uptodeg = 5; int lV = 3;
def R = makeLetterplaceRing(uptodeg);
setring R;
poly f = x(1)*z(2)*y(3) - 2*z(1)*y(2) + 3*x(1);
shiftPoly(f,1);
↦ x(2)*z(3)*y(4)-2*z(2)*y(3)+3*x(2)
shiftPoly(f,2);
↦ x(3)*z(4)*y(5)-2*z(3)*y(4)+3*x(3)
```

7.7.9.8 lpPower

Procedure from library **freegb.lib** (see Section 7.7.9 [freegb_lib], page 495).

Usage: lpPower(f,n); f letterplace polynomial, int n

Return: poly

Assume: basering has a letterplace ring structure

Purpose: compute the letterplace form of f^n

Example:

```
LIB "freegb.lib";
// define a ring in letterplace form as follows:
ring r = 0,(x(1),y(1),x(2),y(2),x(3),y(3),x(4),y(4)),dp;
def R = setLetterplaceAttributes(r,4,2); // supply R with letterplace structure
setring R;
poly a = x(1)*y(2) + y(1); poly b = y(1) - 1;
lpPower(a,2);
↦ x(1)*y(2)*x(3)*y(4)+y(1)*x(2)*y(3)+x(1)*y(2)*y(3)+y(1)*y(2)
lpPower(b,4);
↦ y(1)*y(2)*y(3)*y(4)-4*y(1)*y(2)*y(3)+6*y(1)*y(2)-4*y(1)+1
```

7.7.9.9 lieBracket

Procedure from library **freegb.lib** (see Section 7.7.9 [freegb_lib], page 495).

Usage: lieBracket(a,b[,N]); a,b letterplace polynomials, N an optional integer

Return: poly

Assume: basering has a letterplace ring structure

Purpose: compute the Lie bracket [a,b] = ab - ba between letterplace polynomials

Note: if N>1 is specified, then the left normed bracket [a,[...[a,b]]]] is computed.

Example:

```
LIB "freegb.lib";
ring r = 0,(x(1),y(1),x(2),y(2),x(3),y(3),x(4),y(4)),dp;
def R = setLetterplaceAttributes(r,4,2); // supply R with letterplace structure
setring R;
poly a = x(1)*y(2); poly b = y(1);
lieBracket(a,b);
↦ -y(1)*x(2)*y(3)+x(1)*y(2)*y(3)
lieBracket(x(1),y(1),2);
↦ -y(1)*x(2)*x(3)*x(4)+3*x(1)*y(2)*x(3)*x(4)-3*x(1)*x(2)*y(3)*x(4)+x(1)*x(2\
   )*x(3)*y(4)
```

7.7.9.10 lp2lstr

Procedure from library `freegb.lib` (see Section 7.7.9 [freegb_lib], page 495).

Usage: lp2lstr(K,s); K an ideal, s a ring name

Return: nothing (exports object `@LN` into the ring named s)

Assume: basering has a letterplace ring structure

Purpose: converts letterplace ideal to list of modules

Note: useful as preprocessing to 'lst2str'

Example:

```
LIB "freegb.lib";
intmat A[2][2] = 2, -1, -1, 2; // sl_3 == A_2
ring r = 0,(f1,f2),dp;
def R = makeLetterplaceRing(3);
setring R;
ideal I = serreRelations(A,1);
lp2lstr(I,r);
setring r;
lst2str(@LN,1);
↦ [1]:
↦    f1*f2*f2-2*f2*f1*f2+f2*f2*f1
↦ [2]:
↦    f1*f1*f2-2*f1*f2*f1+f2*f1*f1
```

7.7.9.11 lst2str

Procedure from library `freegb.lib` (see Section 7.7.9 [freegb_lib], page 495).

Usage: lst2str(L[,n]); L a list of modules, n an optional integer

Return: list (of strings)

Purpose: convert a list (of modules) into polynomials in free algebra

Note: if an optional integer is not 0, stars signs are used in multiplication

Example:

```
LIB "freegb.lib";
ring r = 0,(x,y,z),(dp(1),dp(2));
module M = [-1,x,y],[-7,y,y],[3,x,x];
module N = [1,x,y,x,y],[-2,y,x,y,x],[6,x,y,y,x,y];
list L; L[1] = M; L[2] = N;
lst2str(L);
↦ [1]:
↦    -xy-7yy+3xx
↦ [2]:
↦    xyxy-2yxyx+6xyyxy
lst2str(L[1],1);
↦ [1]:
↦    -x*y-7*y*y+3*x*x
```

7.7.9.12 mod2str

Procedure from library `freegb.lib` (see Section 7.7.9 [freegb_lib], page 495).

Usage: mod2str(M[,n]); M a module, n an optional integer

Return: string

Purpose: convert a module into a polynomial in free algebra

Note: if an optional integer is not 0, stars signs are used in multiplication

Example:

```
LIB "freegb.lib";
ring r = 0,(x,y,z),(dp);
module M = [1,x,y,x,y],[-2,y,x,y,x],[6,x,y,y,x,y];
mod2str(M);
↦ xyxy-2yxyx+6xyyxy
mod2str(M,1);
↦ x*y*x*y-2*y*x*y*x+6*x*y*y*x*y
```

7.7.9.13 vct2str

Procedure from library `freegb.lib` (see Section 7.7.9 [freegb_lib], page 495).

Usage: vct2str(v[,n]); v a vector, n an optional integer

Return: string

Purpose: convert a vector into a word in free algebra

Note: if an optional integer is not 0, stars signs are used in multiplication

Example:

```
LIB "freegb.lib";
ring r = (0,a),(x,y3,z(1)),dp;
vector v = [-7,x,y3^4,x2,z(1)^3];
vct2str(v);
↦ -7xy3y3y3y3xxz(1)z(1)z(1)
vct2str(v,1);
↦ -7*x*y3*y3*y3*y3*x*x*z(1)*z(1)*z(1)
vector w = [-7a^5+6a,x,y3,y3,x,z(1),z(1)];
vct2str(w);
```

```
↦ (-16807*a^5+6*a)xy3y3xz(1)z(1)
vct2str(w,1);
↦ (-16807*a^5+6*a)*x*y3*y3*x*z(1)*z(1)
```

7.7.9.14 serreRelations

Procedure from library `freegb.lib` (see Section 7.7.9 [freegb_lib], page 495).

Usage: serreRelations(A,z); A an intmat, z an int

Return: ideal

Assume: basering has a letterplace ring structure and
A is a generalized Cartan matrix with integer entries

Purpose: compute the ideal of Serre's relations associated to A

Example:

```
LIB "freegb.lib";
intmat A[3][3] =
2, -1, 0,
-1, 2, -3,
0, -1, 2; // G^1_2 Cartan matrix
ring r = 0,(f1,f2,f3),dp;
int uptodeg = 5;
def R = makeLetterplaceRing(uptodeg);
setring R;
ideal I = serreRelations(A,1); I = simplify(I,1+2+8);
I;
↦ I[1]=f1(1)*f2(2)*f2(3)-2*f2(1)*f1(2)*f2(3)+f2(1)*f2(2)*f1(3)
↦ I[2]=f1(1)*f3(2)-f3(1)*f1(2)
↦ I[3]=f1(1)*f1(2)*f2(3)-2*f1(1)*f2(2)*f1(3)+f2(1)*f1(2)*f1(3)
↦ I[4]=f2(1)*f3(2)*f3(3)*f3(4)*f3(5)-4*f3(1)*f2(2)*f3(3)*f3(4)*f3(5)+6*f3(1\
   )*f3(2)*f2(3)*f3(4)*f3(5)-4*f3(1)*f3(2)*f3(3)*f2(4)*f3(5)+f3(1)*f3(2)*f3(\
   3)*f3(4)*f2(5)
↦ I[5]=f2(1)*f2(2)*f3(3)-2*f2(1)*f3(2)*f2(3)+f3(1)*f2(2)*f2(3)
```

7.7.9.15 fullSerreRelations

Procedure from library `freegb.lib` (see Section 7.7.9 [freegb_lib], page 495).

Usage: fullSerreRelations(A,N,C,P,d); A an intmat, N,C,P ideals, d an int

Return: ring (and ideal)

Purpose: compute the inhomogeneous Serre's relations associated to A in given
variable names

Assume: three ideals in the input are of the same sizes and contain merely
variables which are interpreted as follows: N resp. P stand for negative
resp. positive roots, C stand for Cartan elements. d is the degree bound for
letterplace ring, which will be returned.
The matrix A is a generalized Cartan matrix with integer entries
The result is the ideal called 'fsRel' in the returned ring.

Example:

```
LIB "freegb.lib";
intmat A[2][2] =
2, -1,
-1, 2; // A_2 = sl_3 Cartan matrix
ring r = 0,(f1,f2,h1,h2,e1,e2),dp;
ideal negroots = f1,f2; ideal cartans = h1,h2; ideal posroots = e1,e2;
int uptodeg = 5;
def RS = fullSerreRelations(A,negroots,cartans,posroots,uptodeg);
setring RS; fsRel;
↦ fsRel[1]=f1(1)*f2(2)*f2(3)-2*f2(1)*f1(2)*f2(3)+f2(1)*f2(2)*f1(3)
↦ fsRel[2]=f1(1)*f1(2)*f2(3)-2*f1(1)*f2(2)*f1(3)+f2(1)*f1(2)*f1(3)
↦ fsRel[3]=e1(1)*e2(2)*e2(3)-2*e2(1)*e1(2)*e2(3)+e2(1)*e2(2)*e1(3)
↦ fsRel[4]=e1(1)*e1(2)*e2(3)-2*e1(1)*e2(2)*e1(3)+e2(1)*e1(2)*e1(3)
↦ fsRel[5]=f2(1)*e1(2)-e1(1)*f2(2)
↦ fsRel[6]=f1(1)*e2(2)-e2(1)*f1(2)
↦ fsRel[7]=-f1(1)*e1(2)+e1(1)*f1(2)-h1(1)
↦ fsRel[8]=-f2(1)*e2(2)+e2(1)*f2(2)-h2(1)
↦ fsRel[9]=h1(1)*h2(2)-h2(1)*h1(2)
↦ fsRel[10]=h1(1)*e1(2)-e1(1)*h1(2)-2*e1(1)
↦ fsRel[11]=f1(1)*h1(2)-h1(1)*f1(2)-2*f1(1)
↦ fsRel[12]=h1(1)*e2(2)-e2(1)*h1(2)+e2(1)
↦ fsRel[13]=f2(1)*h1(2)-h1(1)*f2(2)+f2(1)
↦ fsRel[14]=h2(1)*e1(2)-e1(1)*h2(2)+e1(1)
↦ fsRel[15]=f1(1)*h2(2)-h2(1)*f1(2)+f1(1)
↦ fsRel[16]=h2(1)*e2(2)-e2(1)*h2(2)-2*e2(1)
↦ fsRel[17]=f2(1)*h2(2)-h2(1)*f2(2)-2*f2(1)
```

7.7.9.16 isVar

Procedure from library `freegb.lib` (see Section 7.7.9 [freegb_lib], page 495).

Usage: isVar(p); poly p

Return: int

Purpose: check, whether leading monomial of p is a power of a single variable from the basering. Returns the exponent or 0 if p is multivariate.

Example:

```
LIB "freegb.lib";
ring r = 0,(x,y),dp;
poly f = xy+1;
isVar(f);
↦ 0
poly g = y^3;
isVar(g);
↦ 3
poly h = 7*x^3;
isVar(h);
↦ 3
poly i = 1;
isVar(i);
↦ 0
```

7.7.9.17 ademRelations

Procedure from library `freegb.lib` (see Section 7.7.9 [freegb_lib], page 495).

Usage: ademRelations(i,j); i,j int

Return: ring (and exports ideal)

Assume: there are at least i+j variables in the basering

Purpose: compute the ideal of Adem relations for i<2j in characteristic 0
the ideal is exported under the name AdemRel in the output ring

Example:

```
LIB "freegb.lib";
def A = ademRelations(2,5);
setring A;
AdemRel;
↦ 6*s(7)*s(0)+s(6)*s(1)
```

7.7.10 involut_lib

Library: involut.lib

Purpose: Computations and operations with involutions

Authors: Oleksandr Iena, yena@mathematik.uni-kl.de,
Markus Becker, mbecker@mathematik.uni-kl.de,
Viktor Levandovskyy, levandov@mathematik.uni-kl.de

Overview: Involution is an anti-automorphism of a non-commutative K-algebra with the property that applied an involution twice, one gets an identity. Involution is linear with respect to the ground field. In this library we compute linear involutions, distinguishing the case of a diagonal matrix (such involutions are called homothetic) and a general one. Also, linear automorphisms of different order can be computed.

Support: Forschungsschwerpunkt 'Mathematik und Praxis' (Project of Dr. E. Zerz and V. Levandovskyy), Uni Kaiserslautern

Remark: This library provides algebraic tools for computations and operations with algebraic involutions and linear automorphisms of non-commutative algebras

Procedures:

7.7.10.1 findInvo

Procedure from library `involut.lib` (see Section 7.7.10 [involut_lib], page 505).

Usage: findInvo();

Return: a ring containing a list L of pairs, where
L[i][1] = ideal; a Groebner Basis of an i-th associated prime,
L[i][2] = matrix, defining a linear map, with entries, reduced with respect to L[i][1]

Purpose: computed the ideal of linear involutions of the basering

Assume: the relations on the algebra are of the form YX = XY + D, that is the current ring is a G-algebra of Lie type.

Note: for convenience, the full ideal of relations `idJ` and the initial matrix with indeterminates `matD` are exported in the output ring

Example:

```
LIB "involut.lib";
def a = makeWeyl(1);
setring a; // this algebra is a first Weyl algebra
a;
↦ // coefficients: QQ
↦ // number of vars : 2
↦ //        block   1 : ordering dp
↦ //                  : names    x D
↦ //        block   2 : ordering C
↦ // noncommutative relations:
↦ //    Dx=xD+1
def X = findInvo();
setring X; // ring with new variables, corr. to unknown coefficients
X;
↦ // coefficients: QQ
↦ // number of vars : 4
↦ //        block   1 : ordering dp
↦ //                  : names    a11 a12 a21 a22
↦ //        block   2 : ordering C
L;
↦ [1]:
↦    [1]:
↦       _[1]=a11+a22
↦       _[2]=a12*a21+a22^2-1
↦    [2]:
↦       _[1,1]=-a22
↦       _[1,2]=a12
↦       _[2,1]=a21
↦       _[2,2]=a22
// look at the matrix in the new variables, defining the linear involution
print(L[1][2]);
↦ -a22,a12,
↦ a21, a22
L[1][1];  // where new variables obey these relations
↦ _[1]=a11+a22
↦ _[2]=a12*a21+a22^2-1
idJ;
↦ idJ[1]=-a12*a21+a11*a22+1
↦ idJ[2]=a11^2+a12*a21-1
↦ idJ[3]=a11*a12+a12*a22
↦ idJ[4]=a11*a21+a21*a22
↦ idJ[5]=a12*a21+a22^2-1
```

See also: Section 7.7.10.2 [findInvoDiag], page 506; Section 7.7.10.5 [involution], page 510.

7.7.10.2 findInvoDiag

Procedure from library `involut.lib` (see Section 7.7.10 [involut_lib], page 505).

Usage: findInvoDiag();

Return: a ring together with a list of pairs L, where
L[i][1] = ideal; a Groebner Basis of an i-th associated prime,
L[i][2] = matrix, defining a linear map, with entries, reduced with respect to L[i][1]

Purpose: compute homothetic (diagonal) involutions of the basering

Assume: the relations on the algebra are of the form YX = XY + D, that is the current ring is a G-algebra of Lie type.

Note: for convenience, the full ideal of relations `idJ` and the initial matrix with indeterminates `matD` are exported in the output ring

Example:

```
LIB "involut.lib";
def a = makeWeyl(1);
setring a; // this algebra is a first Weyl algebra
a;
↦ // coefficients: QQ
↦ // number of vars : 2
↦ //        block   1 : ordering dp
↦ //                  : names    x D
↦ //        block   2 : ordering C
↦ // noncommutative relations:
↦ //    Dx=xD+1
def X = findInvoDiag();
setring X; // ring with new variables, corresponding to unknown coefficients
X;
↦ // coefficients: QQ
↦ // number of vars : 2
↦ //        block   1 : ordering dp
↦ //                  : names    a11 a22
↦ //        block   2 : ordering C
// print matrices, defining linear involutions
print(L[1][2]);  // a first matrix: we see it is constant
↦ -1,0,
↦ 0, 1
print(L[2][2]);  // and a second possible matrix; it is constant too
↦ 1,0,
↦ 0,-1
L; // let us take a look on the whole list
↦ [1]:
↦    [1]:
↦       _[1]=a22-1
↦       _[2]=a11+1
↦    [2]:
↦       _[1,1]=-1
↦       _[1,2]=0
↦       _[2,1]=0
↦       _[2,2]=1
↦ [2]:
↦    [1]:
↦       _[1]=a22+1
↦       _[2]=a11-1
↦    [2]:
↦       _[1,1]=1
```

```
↦        _[1,2]=0
↦        _[2,1]=0
↦        _[2,2]=-1
idJ;
↦ idJ[1]=a11*a22+1
↦ idJ[2]=a11^2-1
↦ idJ[3]=a22^2-1
```

See also: Section 7.7.10.1 [findInvo], page 505; Section 7.7.10.5 [involution], page 510.

7.7.10.3 findAuto

Procedure from library `involut.lib` (see Section 7.7.10 [involut_lib], page 505).

Usage: findAuto(n); n an integer

Return: a ring together with a list of pairs L, where
L[i][1] = ideal; a Groebner Basis of an i-th associated prime,
L[i][2] = matrix, defining a linear map, with entries, reduced with respect to L[i][1]

Purpose: compute the ideal of linear automorphisms of the basering,
given by a matrix, n-th power of which gives identity (i.e. unipotent matrix)

Assume: the relations on the algebra are of the form YX = XY + D, that is the current ring is a G-algebra of Lie type.

Note: if n=0, a matrix, defining an automorphism is not assumed to be unipotent
but just non-degenerate. A nonzero parameter `@p` is introduced as the value of
the determinant of the matrix above.
For convenience, the full ideal of relations `idJ` and the initial matrix with indeterminates
`matD` are mutually exported in the output ring

Example:

```
LIB "involut.lib";
def a = makeWeyl(1);
setring a; // this algebra is a first Weyl algebra
a;
↦ // coefficients: QQ
↦ // number of vars : 2
↦ //        block   1 : ordering dp
↦ //                  : names    x D
↦ //        block   2 : ordering C
↦ // noncommutative relations:
↦ //    Dx=xD+1
def X = findAuto(2);  // in contrast to findInvo look for automorphisms
setring X; // ring with new variables - unknown coefficients
X;
↦ // coefficients: QQ
↦ // number of vars : 4
↦ //        block   1 : ordering dp
↦ //                  : names    a11 a12 a21 a22
↦ //        block   2 : ordering C
size(L); // we have (size(L)) families in the answer
↦ 2
// look at matrices, defining linear automorphisms:
```

```
print(L[1][2]);  // a first one: we see it is the identity
↦ 1,0,
↦ 0,1
print(L[2][2]);  // and a second possible matrix; it is diagonal
↦ -1,0,
↦ 0, -1
// L; // we can take a look on the whole list, too
idJ;
↦ idJ[1]=-a12*a21+a11*a22-1
↦ idJ[2]=a11^2+a12*a21-1
↦ idJ[3]=a11*a12+a12*a22
↦ idJ[4]=a11*a21+a21*a22
↦ idJ[5]=a12*a21+a22^2-1
kill X; kill a;
//----------- find all the linear automorphisms --------------------
//----------- use the call findAuto(0)          --------------------
ring R = 0,(x,s),dp;
def r = nc_algebra(1,s); setring r; // the shift algebra
s*x; // the only relation in the algebra is:
↦ xs+s
def Y = findAuto(0);
setring Y;
size(L); // here, we have 1 parametrized family
↦ 1
print(L[1][2]); // here, @p is a nonzero parameter
↦ 1,a12,
↦ 0,(@p)
det(L[1][2]-@p);  // check whether determinante is zero
↦ 0
```

See also: Section 7.7.10.1 [findInvo], page 505.

7.7.10.4 ncdetection

Procedure from library `involut.lib` (see Section 7.7.10 [involut_lib], page 505).

Usage: ncdetection();

Return: ideal, representing an involution map

Purpose: compute classical involutions (i.e. acting rather on operators than on variables) for some particular noncommutative algebras

Assume: the procedure is aimed at non-commutative algebras with differential, shift or advance operators arising in Control Theory. It has to be executed in a ring.

Example:

```
LIB "involut.lib";
ring R = 0,(x,y,z,D(1..3)),dp;
matrix D[6][6];
D[1,4]=1; D[2,5]=1;  D[3,6]=1;
def r = nc_algebra(1,D); setring r;
ncdetection();
↦ _[1]=x
↦ _[2]=y
↦ _[3]=z
```

```
↦ _[4]=-D(1)
↦ _[5]=-D(2)
↦ _[6]=-D(3)
kill r, R;
//----------------------------------------
ring R=0,(x,S),dp;
def r = nc_algebra(1,-S); setring r;
ncdetection();
↦ _[1]=-x
↦ _[2]=S
kill r, R;
//----------------------------------------
ring R=0,(x,D(1),S),dp;
matrix D[3][3];
D[1,2]=1;  D[1,3]=-S;
def r = nc_algebra(1,D); setring r;
ncdetection();
↦ _[1]=-x
↦ _[2]=D(1)
↦ _[3]=S
```

7.7.10.5 involution

Procedure from library `involut.lib` (see Section 7.7.10 [involut_lib], page 505).

Usage: involution(m, theta); m is a poly/vector/ideal/matrix/module, theta is a map

Return: object of the same type as m

Purpose: applies the involution, presented by theta to the object m

Theory: for an involution theta and two polynomials a,b from the algebra,
theta(ab) = theta(b) theta(a); theta is linear with respect to the ground field

Note: This is generalized "theta(m)" for data types unsupported by "map".

Example:

```
LIB "involut.lib";
ring R = 0,(x,d),dp;
def r = nc_algebra(1,1); setring r; // Weyl-Algebra
map F = r,x,-d;
F(F);  // should be maxideal(1) for an involution
↦ _[1]=x
↦ _[2]=d
poly f =  x*d^2+d;
poly If = involution(f,F);
f-If;
↦ 0
poly g = x^2*d+2*x*d+3*x+7*d;
poly tg = -d*x^2-2*d*x+3*x-7*d;
poly Ig = involution(g,F);
tg-Ig;
↦ 0
ideal I = f,g;
ideal II = involution(I,F);
II;
```

```
↦ II[1]=xd2+d
↦ II[2]=-x2d-2xd+x-7d-2
matrix(I) - involution(II,F);
↦ _[1,1]=0
↦ _[1,2]=0
module M  = [f,g,0],[g,0,x^2*d];
module IM = involution(M,F);
print(IM);
↦ xd2+d,          -x2d-2xd+x-7d-2,
↦ -x2d-2xd+x-7d-2,0,
↦ 0,              -x2d-2x
print(matrix(M) - involution(IM,F));
↦ 0,0,
↦ 0,0,
↦ 0,0
```

7.7.10.6 isInvolution

Procedure from library `involut.lib` (see Section 7.7.10 [involut_lib], page 505).

Usage: isInvolution(F); F is a map from current ring to itself

Return: integer, 1 if F determines an involution and 0 otherwise

Theory: involution is an antiautomorphism of order 2

Assume: F is a map from current ring to itself

Example:

```
LIB "involut.lib";
def A = makeUsl(2); setring A;
map I = A,-e,-f,-h; //correct antiauto involution
isInvolution(I);
↦ 1
map J = A,3*e,1/3*f,-h; // antiauto but not involution
isInvolution(J);
↦ 0
map K = A,f,e,-h; // not antiauto
isInvolution(K);
↦ 0
```

See also: Section 7.7.10.1 [findInvo], page 505; Section 7.7.10.5 [involution], page 510; Section 7.7.10.7 [isAntiEndo], page 511.

7.7.10.7 isAntiEndo

Procedure from library `involut.lib` (see Section 7.7.10 [involut_lib], page 505).

Usage: isAntiEndo(F); F is a map from current ring to itself

Return: integer, 1 if F determines an antiendomorphism of
current ring and 0 otherwise

Assume: F is a map from current ring to itself

Example:

```
LIB "involut.lib";
def A = makeUsl(2); setring A;
map I = A,-e,-f,-h; //correct antiauto involution
isAntiEndo(I);
↦ 1
map J = A,3*e,1/3*f,-h; // antiauto but not involution
isAntiEndo(J);
↦ 1
map K = A,f,e,-h; // not antiendo
isAntiEndo(K);
↦ 0
```

See also: Section 7.7.10.1 [findInvo], page 505; Section 7.7.10.5 [involution], page 510; Section 7.7.10.6 [isInvolution], page 511.

7.7.11 gkdim_lib

Library: gkdim.lib

Purpose: Procedures for calculating the Gelfand-Kirillov dimension

Authors: Lobillo, F.J., jlobillo@ugr.es
Rabelo, C., crabelo@ugr.es

Support: 'Metodos algebraicos y efectivos en grupos cuanticos', BFM2001-3141, MCYT, Jose Gomez-Torrecillas (Main researcher).

Note: The built-in command `dim`, executed for a module in @plural, computes the Gelfand-Kirillov dimension.

Procedures:

7.7.11.1 GKdim

Procedure from library `gkdim.lib` (see Section 7.7.11 [gkdim_lib], page 512).

Usage: GKdim(L); L is a left ideal/module/matrix

Return: int

Purpose: compute the Gelfand-Kirillov dimension of the factor-module, whose presentation is given by L, e.g. R^r/L

Note: if the factor-module is zero, -1 is returned

Example:

```
LIB "gkdim.lib";
ring R = 0,(x,y,z),Dp;
matrix C[3][3]=0,1,1,0,0,-1,0,0,0;
matrix D[3][3]=0,0,0,0,0,x;
def r = nc_algebra(C,D); setring r;
r;
↦ // coefficients: QQ
↦ // number of vars : 3
↦ //        block   1 : ordering Dp
↦ //                  : names    x y z
↦ //        block   2 : ordering C
↦ // noncommutative relations:
```

```
↦ //    zy=-yz+x
ideal I=x;
GKdim(I);
↦ 2
ideal J=x2,y;
GKdim(J);
↦ 1
module M=[x2,y,1],[x,y2,0];
GKdim(M);
↦ 3
ideal A = x,y,z;
GKdim(A);
↦ 0
ideal B = 1;
GKdim(B);
↦ -1
GKdim(ideal(0)) == nvars(basering);  // should be true, i.e., evaluated to 1
↦ 1
```

7.7.12 ncalg_lib

Library: ncalg.lib

Purpose: Definitions of important G- and GR-algebras

Authors: Viktor Levandovskyy, levandov@mathematik.uni-kl.de,
Oleksandr Motsak, U@D, where U={motsak}, D={mathematik.uni-kl.de}

Conventions:
This library provides pre-defined important noncommutative algebras.
For universal enveloping algebras of finite dimensional Lie algebras sl_n, gl_n, g_2 etc. there are functions `makeUsl`, `makeUgl`, `makeUg2` etc.
For quantized enveloping algebras U_q(sl_2) and U_q(sl_3), there are functions `makeQsl2`, `makeQsl3`) and for non-standard quantum deformation of so_3, there is the function `makeQso3`.
For bigger algebras we suppress the output of the (lengthy) list of non-commutative relations and provide only the number of these relations instead.

Procedures:

7.7.12.1 makeUsl2

Procedure from library `ncalg.lib` (see Section 7.7.12 [ncalg_lib], page 513).

Usage: makeUsl2([p]), p an optional integer (field characteristic)

Return: ring

Purpose: set up the U(sl_2) in the variables e,f,h over the field of char p

Note: activate this ring with the `setring` command

Example:

```
LIB "ncalg.lib";
def a=makeUsl2();
setring a;
a;
```

```
↦ // coefficients: QQ
↦ // number of vars : 3
↦ //        block   1 : ordering dp
↦ //                  : names    e f h
↦ //        block   2 : ordering C
↦ // noncommutative relations:
↦ //    fe=ef-h
↦ //    he=eh+2e
↦ //    hf=fh-2f
```

See also: Section 7.7.12.17 [makeUg2], page 523; Section 7.7.12.3 [makeUgl], page 515; Section 7.7.12.2 [makeUsl], page 514.

7.7.12.2 makeUsl

Procedure from library `ncalg.lib` (see Section 7.7.12 [ncalg_lib], page 513).

Usage: makeUsl(n,[p]); n an integer, n>1; p an optional integer (field characteristic)

Return: ring

Purpose: set up the U(sl_n) in the variables (x(i),y(i),h(i) | i=1..n+1) over the field of char p

Note: activate this ring with the `setring` command
This presentation of U(sl_n) is the standard one, i.e. positive resp. negative roots are denoted by x(i) resp. y(i) and the Cartan elements are denoted by h(i).
The variables are ordered as x(1),...x(n),y(1),...,y(n),h(1),...h(n).

Example:

```
LIB "ncalg.lib";
def a=makeUsl(3);
setring a;
a;
↦ // coefficients: QQ
↦ // number of vars : 8
↦ //        block   1 : ordering dp
↦ //                  : names    x(1) x(2) x(3) y(1) y(2) y(3) h(1) h(2)
↦ //        block   2 : ordering C
↦ // noncommutative relations:
↦ //    x(2)x(1)=x(1)*x(2)+x(3)
↦ //    y(1)x(1)=x(1)*y(1)-h(1)
↦ //    y(3)x(1)=x(1)*y(3)-y(2)
↦ //    h(1)x(1)=x(1)*h(1)+2*x(1)
↦ //    h(2)x(1)=x(1)*h(2)-x(1)
↦ //    y(2)x(2)=x(2)*y(2)-h(2)
↦ //    y(3)x(2)=x(2)*y(3)+y(1)
↦ //    h(1)x(2)=x(2)*h(1)-x(2)
↦ //    h(2)x(2)=x(2)*h(2)+2*x(2)
↦ //    y(1)x(3)=x(3)*y(1)-x(2)
↦ //    y(2)x(3)=x(3)*y(2)+x(1)
↦ //    y(3)x(3)=x(3)*y(3)-h(1)-h(2)
↦ //    h(1)x(3)=x(3)*h(1)+x(3)
↦ //    h(2)x(3)=x(3)*h(2)+x(3)
↦ //    y(2)y(1)=y(1)*y(2)-y(3)
↦ //    h(1)y(1)=y(1)*h(1)-2*y(1)
↦ //    h(2)y(1)=y(1)*h(2)+y(1)
```

```
↦ //    h(1)y(2)=y(2)*h(1)+y(2)
↦ //    h(2)y(2)=y(2)*h(2)-2*y(2)
↦ //    h(1)y(3)=y(3)*h(1)-y(3)
↦ //    h(2)y(3)=y(3)*h(2)-y(3)
```

See also: Section 7.7.12.24 [makeQsl3], page 528; Section 7.7.12.22 [makeQso3], page 527; Section 7.7.12.17 [makeUg2], page 523; Section 7.7.12.3 [makeUgl], page 515; Section 7.7.12.1 [makeUsl2], page 513.

7.7.12.3 makeUgl

Procedure from library `ncalg.lib` (see Section 7.7.12 [ncalg_lib], page 513).

Usage: makeUgl(n,[p]); n an int, n>1; p an optional int (field characteristic)

Return: ring

Purpose: set up the U(gl_n) in the (e_ij (1<i,j<n)) presentation (where e_ij corresponds to a matrix with 1 at i,j only) over the field of char p

Note: activate this ring with the `setring` command
the variables are ordered as e_12,e_13,...,e_1n,e_21,...,e_nn.

Example:

```
LIB "ncalg.lib";
def a=makeUgl(3);
setring a; a;
↦ // coefficients: QQ
↦ // number of vars : 9
↦ //        block   1 : ordering dp
↦ //                  : names    e_1_1 e_1_2 e_1_3 e_2_1 e_2_2 e_2_3 e_3_1 \
   e_3_2 e_3_3
↦ //        block   2 : ordering C
↦ // noncommutative relations:
↦ //    e_1_2e_1_1=e_1_1*e_1_2-e_1_2
↦ //    e_1_3e_1_1=e_1_1*e_1_3-e_1_3
↦ //    e_2_1e_1_1=e_1_1*e_2_1+e_2_1
↦ //    e_3_1e_1_1=e_1_1*e_3_1+e_3_1
↦ //    e_2_1e_1_2=e_1_2*e_2_1-e_1_1+e_2_2
↦ //    e_2_2e_1_2=e_1_2*e_2_2-e_1_2
↦ //    e_2_3e_1_2=e_1_2*e_2_3-e_1_3
↦ //    e_3_1e_1_2=e_1_2*e_3_1+e_3_2
↦ //    e_2_1e_1_3=e_1_3*e_2_1+e_2_3
↦ //    e_3_1e_1_3=e_1_3*e_3_1-e_1_1+e_3_3
↦ //    e_3_2e_1_3=e_1_3*e_3_2-e_1_2
↦ //    e_3_3e_1_3=e_1_3*e_3_3-e_1_3
↦ //    e_2_2e_2_1=e_2_1*e_2_2+e_2_1
↦ //    e_3_2e_2_1=e_2_1*e_3_2+e_3_1
↦ //    e_2_3e_2_2=e_2_2*e_2_3-e_2_3
↦ //    e_3_2e_2_2=e_2_2*e_3_2+e_3_2
↦ //    e_3_1e_2_3=e_2_3*e_3_1-e_2_1
↦ //    e_3_2e_2_3=e_2_3*e_3_2-e_2_2+e_3_3
↦ //    e_3_3e_2_3=e_2_3*e_3_3-e_2_3
↦ //    e_3_3e_3_1=e_3_1*e_3_3+e_3_1
↦ //    e_3_3e_3_2=e_3_2*e_3_3+e_3_2
```

See also: Section 7.7.12.17 [makeUg2], page 523; Section 7.7.12.2 [makeUsl], page 514.

7.7.12.4 makeUso5

Procedure from library `ncalg.lib` (see Section 7.7.12 [ncalg_lib], page 513).

Usage: makeUso5([p]); p an optional integer (field characteristic)

Return: a ring, describing U(so_5)

Note: You have to activate this ring with the 'setring' command. The presentation of U(so_5) is derived from the Chevalley representation of so_5, positive resp. negative roots are denoted by x(i) resp. y(i); Cartan elements are denoted by h(i).

Example:

```
LIB "ncalg.lib";
def ncAlgebra = makeUso5();
ncAlgebra;
↦ // coefficients: QQ
↦ // number of vars : 10
↦ //        block   1 : ordering dp
↦ //                  : names    X(1) X(2) X(3) X(4) Y(1) Y(2) Y(3) Y(4) H(\
   1) H(2)
↦ //        block   2 : ordering C
↦ // noncommutative relations: ...
setring ncAlgebra;
// ...  28  noncommutative relations
```

See also: Section 7.7.12.19 [makeUe6], page 524; Section 7.7.12.20 [makeUe7], page 525; Section 7.7.12.21 [makeUe8], page 526; Section 7.7.12.18 [makeUf4], page 524; Section 7.7.12.17 [makeUg2], page 523; Section 7.7.12.2 [makeUsl], page 514; Section 7.7.12.12 [makeUsp1], page 520.

7.7.12.5 makeUso6

Procedure from library `ncalg.lib` (see Section 7.7.12 [ncalg_lib], page 513).

Usage: makeUso6([p]); p an optional integer (field characteristic)

Return: a ring, describing U(so_6)

Note: You have to activate this ring with the 'setring' command. The presentation of U(so_6) is derived from the Chevalley representation of so_6, positive resp. negative roots are denoted by x(i) resp. y(i); Cartan elements are denoted by h(i).

Example:

```
LIB "ncalg.lib";
def ncAlgebra = makeUso6();
ncAlgebra;
↦ // coefficients: QQ
↦ // number of vars : 15
↦ //        block   1 : ordering dp
↦ //                  : names    X(1) X(2) X(3) X(4) X(5) X(6) Y(1) Y(2) Y(\
   3) Y(4) Y(5) Y(6) H(1) H(2) H(3)
↦ //        block   2 : ordering C
↦ // noncommutative relations: ...
setring ncAlgebra;
// ...  60  noncommutative relations
```

See also: Section 7.7.12.19 [makeUe6], page 524; Section 7.7.12.20 [makeUe7], page 525; Section 7.7.12.21 [makeUe8], page 526; Section 7.7.12.18 [makeUf4], page 524; Section 7.7.12.17

[makeUg2], page 523; Section 7.7.12.2 [makeUsl], page 514; Section 7.7.12.4 [makeUso5], page 516; Section 7.7.12.12 [makeUsp1], page 520.

7.7.12.6 makeUso7

Procedure from library `ncalg.lib` (see Section 7.7.12 [ncalg_lib], page 513).

Usage: makeUso7([p]); p an optional integer (field characteristic)

Return: a ring, describing U(so_7)

Note: You have to activate this ring with the 'setring' command. The presentation of U(so_7) is derived from the Chevalley representation of so_7, positive resp. negative roots are denoted by x(i) resp. y(i); Cartan elements are denoted by h(i).

Example:

```
LIB "ncalg.lib";
def ncAlgebra = makeUso7();
ncAlgebra;
↦ // coefficients: QQ
↦ // number of vars : 21
↦ //        block   1 : ordering dp
↦ //                  : names    X(1) X(2) X(3) X(4) X(5) X(6) X(7) X(8) X(\
   9) Y(1) Y(2) Y(3) Y(4) Y(5) Y(6) Y(7) Y(8) Y(9) H(1) H(2) H(3)
↦ //        block   2 : ordering C
↦ // noncommutative relations: ...
setring ncAlgebra;
// ...  107  noncommutative relations
```

See also: Section 7.7.12.19 [makeUe6], page 524; Section 7.7.12.20 [makeUe7], page 525; Section 7.7.12.21 [makeUe8], page 526; Section 7.7.12.18 [makeUf4], page 524; Section 7.7.12.17 [makeUg2], page 523; Section 7.7.12.2 [makeUsl], page 514; Section 7.7.12.4 [makeUso5], page 516; Section 7.7.12.12 [makeUsp1], page 520.

7.7.12.7 makeUso8

Procedure from library `ncalg.lib` (see Section 7.7.12 [ncalg_lib], page 513).

Usage: makeUso8([p]); p an optional integer (field characteristic)

Return: a ring, describing U(so_8)

Note: You have to activate this ring with the 'setring' command. The presentation of U(so_8) is derived from the Chevalley representation of so_8, positive resp. negative roots are denoted by x(i) resp. y(i); Cartan elements are denoted by h(i).

Example:

```
LIB "ncalg.lib";
def ncAlgebra = makeUso8();
ncAlgebra;
↦ // coefficients: QQ
↦ // number of vars : 28
↦ //        block   1 : ordering dp
↦ //                  : names    X(1) X(2) X(3) X(4) X(5) X(6) X(7) X(8) X(\
   9) X(10) X(11) X(12) Y(1) Y(2) Y(3) Y(4) Y(5) Y(6) Y(7) Y(8) Y(9) Y(10) Y\
   (11) Y(12) H(1) H(2) H(3) H(4)
↦ //        block   2 : ordering C
```

```
↦ // noncommutative relations: ...
setring ncAlgebra;
// ...  180  noncommutative relations
```

See also: Section 7.7.12.19 [makeUe6], page 524; Section 7.7.12.20 [makeUe7], page 525; Section 7.7.12.21 [makeUe8], page 526; Section 7.7.12.18 [makeUf4], page 524; Section 7.7.12.17 [makeUg2], page 523; Section 7.7.12.2 [makeUsl], page 514; Section 7.7.12.4 [makeUso5], page 516; Section 7.7.12.12 [makeUsp1], page 520.

7.7.12.8 makeUso9

Procedure from library `ncalg.lib` (see Section 7.7.12 [ncalg_lib], page 513).

Usage: makeUso9([p]); p an optional integer (field characteristic)

Return: a ring, describing U(so_9)

Note: You have to activate this ring with the 'setring' command. The presentation of U(so_9) is derived from the Chevalley representation of so_9, positive resp. negative roots are denoted by x(i) resp. y(i); Cartan elements are denoted by h(i).

Example:

```
LIB "ncalg.lib";
def ncAlgebra = makeUso9();
ncAlgebra;
↦ // coefficients: QQ
↦ // number of vars : 36
↦ //        block   1 : ordering dp
↦ //                  : names    X(1) X(2) X(3) X(4) X(5) X(6) X(7) X(8) X(\
   9) X(10) X(11) X(12) X(13) X(14) X(15) X(16) Y(1) Y(2) Y(3) Y(4) Y(5) Y(6\
   ) Y(7) Y(8) Y(9) Y(10) Y(11) Y(12) Y(13) Y(14) Y(15) Y(16) H(1) H(2) H(3)\
    H(4)
↦ //        block   2 : ordering C
↦ // noncommutative relations: ...
setring ncAlgebra;
// ...  264  noncommutative relations
```

See also: Section 7.7.12.19 [makeUe6], page 524; Section 7.7.12.20 [makeUe7], page 525; Section 7.7.12.21 [makeUe8], page 526; Section 7.7.12.18 [makeUf4], page 524; Section 7.7.12.17 [makeUg2], page 523; Section 7.7.12.2 [makeUsl], page 514; Section 7.7.12.4 [makeUso5], page 516; Section 7.7.12.12 [makeUsp1], page 520.

7.7.12.9 makeUso10

Procedure from library `ncalg.lib` (see Section 7.7.12 [ncalg_lib], page 513).

Usage: makeUso10([p]); p an optional integer (field characteristic)

Return: a ring, describing U(so_{10})

Note: You have to activate this ring with the 'setring' command. The presentation of U(so_{10}) is derived from the Chevalley representation of so_{10}, positive resp. negative roots are denoted by x(i) resp. y(i); Cartan elements are denoted by h(i).

Example:

```
LIB "ncalg.lib";
def ncAlgebra = makeUso10();
```

```
ncAlgebra;
↦ // coefficients: QQ
↦ // number of vars : 45
↦ //        block   1 : ordering dp
↦ //                  : names    X(1) X(2) X(3) X(4) X(5) X(6) X(7) X(8) X(\
   9) X(10) X(11) X(12) X(13) X(14) X(15) X(16) X(17) X(18) X(19) X(20) Y(1)\
    Y(2) Y(3) Y(4) Y(5) Y(6) Y(7) Y(8) Y(9) Y(10) Y(11) Y(12) Y(13) Y(14) Y(\
   15) Y(16) Y(17) Y(18) Y(19) Y(20) H(1) H(2) H(3) H(4) H(5)
↦ //        block   2 : ordering C
↦ // noncommutative relations: ...
setring ncAlgebra;
// ...  390  noncommutative relations
```

See also: Section 7.7.12.19 [makeUe6], page 524; Section 7.7.12.20 [makeUe7], page 525; Section 7.7.12.21 [makeUe8], page 526; Section 7.7.12.18 [makeUf4], page 524; Section 7.7.12.17 [makeUg2], page 523; Section 7.7.12.2 [makeUsl], page 514; Section 7.7.12.4 [makeUso5], page 516; Section 7.7.12.12 [makeUsp1], page 520.

7.7.12.10 makeUso11

Procedure from library `ncalg.lib` (see Section 7.7.12 [ncalg_lib], page 513).

Usage: makeUso11([p]); p an optional integer (field characteristic)

Return: a ring, describing U(so_{11})

Note: You have to activate this ring with the 'setring' command. The presentation of U(so_{11}) is derived from the Chevalley representation of so_{11}, positive resp. negative roots are denoted by x(i) resp. y(i); Cartan elements are denoted by h(i).

Example:

```
LIB "ncalg.lib";
def ncAlgebra = makeUso11();
ncAlgebra;
↦ // coefficients: QQ
↦ // number of vars : 55
↦ //        block   1 : ordering dp
↦ //                  : names    X(1) X(2) X(3) X(4) X(5) X(6) X(7) X(8) X(\
   9) X(10) X(11) X(12) X(13) X(14) X(15) X(16) X(17) X(18) X(19) X(20) X(21\
   ) X(22) X(23) X(24) X(25) Y(1) Y(2) Y(3) Y(4) Y(5) Y(6) Y(7) Y(8) Y(9) Y(\
   10) Y(11) Y(12) Y(13) Y(14) Y(15) Y(16) Y(17) Y(18) Y(19) Y(20) Y(21) Y(2\
   2) Y(23) Y(24) Y(25) H(1) H(2) H(3) H(4) H(5)
↦ //        block   2 : ordering C
↦ // noncommutative relations: ...
setring ncAlgebra;
// ...  523  noncommutative relations
```

See also: Section 7.7.12.19 [makeUe6], page 524; Section 7.7.12.20 [makeUe7], page 525; Section 7.7.12.21 [makeUe8], page 526; Section 7.7.12.18 [makeUf4], page 524; Section 7.7.12.17 [makeUg2], page 523; Section 7.7.12.2 [makeUsl], page 514; Section 7.7.12.4 [makeUso5], page 516; Section 7.7.12.12 [makeUsp1], page 520.

7.7.12.11 makeUso12

Procedure from library `ncalg.lib` (see Section 7.7.12 [ncalg_lib], page 513).

Usage: makeUso12([p]); p an optional integer (field characteristic)

Return: a ring, describing U(so_{12})

Note: You have to activate this ring with the 'setring' command. The presentation of U(so_{12}) is derived from the Chevalley representation of so_{12}, positive resp. negative roots are denoted by x(i) resp. y(i); Cartan elements are denoted by h(i).

Example:

```
LIB "ncalg.lib";
def ncAlgebra = makeUso12();
ncAlgebra;
↦ // coefficients: QQ
↦ // number of vars : 66
↦ //        block   1 : ordering dp
↦ //                  : names    X(1) X(2) X(3) X(4) X(5) X(6) X(7) X(8) X(\
   9) X(10) X(11) X(12) X(13) X(14) X(15) X(16) X(17) X(18) X(19) X(20) X(21\
   ) X(22) X(23) X(24) X(25) X(26) X(27) X(28) X(29) X(30) Y(1) Y(2) Y(3) Y(\
   4) Y(5) Y(6) Y(7) Y(8) Y(9) Y(10) Y(11) Y(12) Y(13) Y(14) Y(15) Y(16) Y(1\
   7) Y(18) Y(19) Y(20) Y(21) Y(22) Y(23) Y(24) Y(25) Y(26) Y(27) Y(28) Y(29\
   ) Y(30) H(1) H(2) H(3) H(4) H(5) H(6)
↦ //        block   2 : ordering C
↦ // noncommutative relations: ...
setring ncAlgebra;
// ...  714  noncommutative relations
```

See also: Section 7.7.12.19 [makeUe6], page 524; Section 7.7.12.20 [makeUe7], page 525; Section 7.7.12.21 [makeUe8], page 526; Section 7.7.12.18 [makeUf4], page 524; Section 7.7.12.17 [makeUg2], page 523; Section 7.7.12.2 [makeUsl], page 514; Section 7.7.12.4 [makeUso5], page 516; Section 7.7.12.12 [makeUsp1], page 520.

7.7.12.12 makeUsp1

Procedure from library `ncalg.lib` (see Section 7.7.12 [ncalg_lib], page 513).

Usage: makeUsp1([p]); p an optional integer (field characteristic)

Return: a ring, describing U(sp_1)

Note: You have to activate this ring with the 'setring' command. The presentation of U(sp_1) is derived from the Chevalley representation of sp_1, positive resp. negative roots are denoted by x(i) resp. y(i); Cartan elements are denoted by h(i).

Example:

```
LIB "ncalg.lib";
def ncAlgebra = makeUsp1();
setring ncAlgebra;
ncAlgebra;
↦ // coefficients: QQ
↦ // number of vars : 3
↦ //        block   1 : ordering dp
↦ //                  : names    X(1) Y(1) H(1)
↦ //        block   2 : ordering C
↦ // noncommutative relations:
↦ //    Y(1)X(1)=X(1)*Y(1)-H(1)
↦ //    H(1)X(1)=X(1)*H(1)+2*X(1)
↦ //    H(1)Y(1)=Y(1)*H(1)-2*Y(1)
```

See also: Section 7.7.12.19 [makeUe6], page 524; Section 7.7.12.20 [makeUe7], page 525; Section 7.7.12.21 [makeUe8], page 526; Section 7.7.12.18 [makeUf4], page 524; Section 7.7.12.17 [makeUg2], page 523; Section 7.7.12.2 [makeUsl], page 514; Section 7.7.12.4 [makeUso5], page 516.

7.7.12.13 makeUsp2

Procedure from library `ncalg.lib` (see Section 7.7.12 [ncalg_lib], page 513).

Usage: makeUsp2([p]); p an optional integer (field characteristic)

Return: a ring, describing U(sp_2)

Note: You have to activate this ring with the 'setring' command. The presentation of U(sp_2) is derived from the Chevalley representation of sp_2, positive resp. negative roots are denoted by x(i) resp. y(i); Cartan elements are denoted by h(i).

Example:

```
LIB "ncalg.lib";
def ncAlgebra = makeUsp2();
setring ncAlgebra;
ncAlgebra;
↦ // coefficients: QQ
↦ // number of vars : 10
↦ //        block   1 : ordering dp
↦ //                  : names    X(1) X(2) X(3) X(4) Y(1) Y(2) Y(3) Y(4) H(\
   1) H(2)
↦ //        block   2 : ordering C
↦ // noncommutative relations:
↦ //    X(2)X(1)=X(1)*X(2)+X(3)
↦ //    X(3)X(1)=X(1)*X(3)+2*X(4)
↦ //    Y(1)X(1)=X(1)*Y(1)-H(1)
↦ //    Y(3)X(1)=X(1)*Y(3)-2*Y(2)
↦ //    Y(4)X(1)=X(1)*Y(4)-Y(3)
↦ //    H(1)X(1)=X(1)*H(1)+2*X(1)
↦ //    H(2)X(1)=X(1)*H(2)-X(1)
↦ //    Y(2)X(2)=X(2)*Y(2)-H(2)
↦ //    Y(3)X(2)=X(2)*Y(3)+Y(1)
↦ //    H(1)X(2)=X(2)*H(1)-2*X(2)
↦ //    H(2)X(2)=X(2)*H(2)+2*X(2)
↦ //    Y(1)X(3)=X(3)*Y(1)-2*X(2)
↦ //    Y(2)X(3)=X(3)*Y(2)+X(1)
↦ //    Y(3)X(3)=X(3)*Y(3)-H(1)-2*H(2)
↦ //    Y(4)X(3)=X(3)*Y(4)+Y(1)
↦ //    H(2)X(3)=X(3)*H(2)+X(3)
↦ //    Y(1)X(4)=X(4)*Y(1)-X(3)
↦ //    Y(3)X(4)=X(4)*Y(3)+X(1)
↦ //    Y(4)X(4)=X(4)*Y(4)-H(1)-H(2)
↦ //    H(1)X(4)=X(4)*H(1)+2*X(4)
↦ //    Y(2)Y(1)=Y(1)*Y(2)-Y(3)
↦ //    Y(3)Y(1)=Y(1)*Y(3)-2*Y(4)
↦ //    H(1)Y(1)=Y(1)*H(1)-2*Y(1)
↦ //    H(2)Y(1)=Y(1)*H(2)+Y(1)
↦ //    H(1)Y(2)=Y(2)*H(1)+2*Y(2)
↦ //    H(2)Y(2)=Y(2)*H(2)-2*Y(2)
↦ //    H(2)Y(3)=Y(3)*H(2)-Y(3)
```

```
↦ //    H(1)Y(4)=Y(4)*H(1)-2*Y(4)
```

See also: Section 7.7.12.19 [makeUe6], page 524; Section 7.7.12.20 [makeUe7], page 525; Section 7.7.12.21 [makeUe8], page 526; Section 7.7.12.18 [makeUf4], page 524; Section 7.7.12.17 [makeUg2], page 523; Section 7.7.12.2 [makeUsl], page 514; Section 7.7.12.4 [makeUso5], page 516; Section 7.7.12.12 [makeUsp1], page 520.

7.7.12.14 makeUsp3

Procedure from library `ncalg.lib` (see Section 7.7.12 [ncalg_lib], page 513).

Usage: makeUsp3([p]); p an optional integer (field characteristic)

Return: a ring, describing U(sp_3)

Note: You have to activate this ring with the 'setring' command. The presentation of U(sp_3) is derived from the Chevalley representation of sp_3, positive resp. negative roots are denoted by x(i) resp. y(i); Cartan elements are denoted by h(i).

Example:

```
LIB "ncalg.lib";
def ncAlgebra = makeUsp3();
ncAlgebra;
↦ // coefficients: QQ
↦ // number of vars : 21
↦ //        block   1 : ordering dp
↦ //                  : names    X(1) X(2) X(3) X(4) X(5) X(6) X(7) X(8) X(\
   9) Y(1) Y(2) Y(3) Y(4) Y(5) Y(6) Y(7) Y(8) Y(9) H(1) H(2) H(3)
↦ //        block   2 : ordering C
↦ // noncommutative relations: ...
setring ncAlgebra;
// ...  107  noncommutative relations
```

See also: Section 7.7.12.19 [makeUe6], page 524; Section 7.7.12.20 [makeUe7], page 525; Section 7.7.12.21 [makeUe8], page 526; Section 7.7.12.18 [makeUf4], page 524; Section 7.7.12.17 [makeUg2], page 523; Section 7.7.12.2 [makeUsl], page 514; Section 7.7.12.4 [makeUso5], page 516; Section 7.7.12.12 [makeUsp1], page 520.

7.7.12.15 makeUsp4

Procedure from library `ncalg.lib` (see Section 7.7.12 [ncalg_lib], page 513).

Usage: makeUsp4([p]); p an optional integer (field characteristic)

Return: a ring, describing U(sp_4)

Note: You have to activate this ring with the 'setring' command. The presentation of U(sp_4) is derived from the Chevalley representation of sp_4, positive resp. negative roots are denoted by x(i) resp. y(i); Cartan elements are denoted by h(i).

Example:

```
LIB "ncalg.lib";
def ncAlgebra = makeUsp4();
ncAlgebra;
↦ // coefficients: QQ
↦ // number of vars : 36
↦ //        block   1 : ordering dp
```

```
↦ //                  : names    X(1) X(2) X(3) X(4) X(5) X(6) X(7) X(8) X(\
   9) X(10) X(11) X(12) X(13) X(14) X(15) X(16) Y(1) Y(2) Y(3) Y(4) Y(5) Y(6\
   ) Y(7) Y(8) Y(9) Y(10) Y(11) Y(12) Y(13) Y(14) Y(15) Y(16) H(1) H(2) H(3)\
    H(4)
↦ //        block   2 : ordering C
↦ // noncommutative relations: ...
setring ncAlgebra;
// ...  264  noncommutative relations
```

See also: Section 7.7.12.19 [makeUe6], page 524; Section 7.7.12.20 [makeUe7], page 525; Section 7.7.12.21 [makeUe8], page 526; Section 7.7.12.18 [makeUf4], page 524; Section 7.7.12.17 [makeUg2], page 523; Section 7.7.12.2 [makeUsl], page 514; Section 7.7.12.4 [makeUso5], page 516; Section 7.7.12.12 [makeUsp1], page 520.

7.7.12.16 makeUsp5

Procedure from library `ncalg.lib` (see Section 7.7.12 [ncalg_lib], page 513).

Usage: makeUsp5([p]); p an optional integer (field characteristic)

Return: a ring, describing U(sp_5)

Note: You have to activate this ring with the 'setring' command. The presentation of U(sp_5) is derived from the Chevalley representation of sp_5, positive resp. negative roots are denoted by x(i) resp. y(i); Cartan elements are denoted by h(i).

Example:

```
LIB "ncalg.lib";
def ncAlgebra = makeUsp5();
ncAlgebra;
↦ // coefficients: QQ
↦ // number of vars : 55
↦ //        block   1 : ordering dp
↦ //                  : names    X(1) X(2) X(3) X(4) X(5) X(6) X(7) X(8) X(\
   9) X(10) X(11) X(12) X(13) X(14) X(15) X(16) X(17) X(18) X(19) X(20) X(21\
   ) X(22) X(23) X(24) X(25) Y(1) Y(2) Y(3) Y(4) Y(5) Y(6) Y(7) Y(8) Y(9) Y(\
   10) Y(11) Y(12) Y(13) Y(14) Y(15) Y(16) Y(17) Y(18) Y(19) Y(20) Y(21) Y(2\
   2) Y(23) Y(24) Y(25) H(1) H(2) H(3) H(4) H(5)
↦ //        block   2 : ordering C
↦ // noncommutative relations: ...
setring ncAlgebra;
// ...  523  noncommutative relations
```

See also: Section 7.7.12.19 [makeUe6], page 524; Section 7.7.12.20 [makeUe7], page 525; Section 7.7.12.21 [makeUe8], page 526; Section 7.7.12.18 [makeUf4], page 524; Section 7.7.12.17 [makeUg2], page 523; Section 7.7.12.2 [makeUsl], page 514; Section 7.7.12.4 [makeUso5], page 516; Section 7.7.12.12 [makeUsp1], page 520.

7.7.12.17 makeUg2

Procedure from library `ncalg.lib` (see Section 7.7.12 [ncalg_lib], page 513).

Usage: makeUg2([p]), p an optional int (field characteristic)

Return: ring

Purpose: set up the U(g_2) in variables (x(i),y(i),Ha,Hb) for i=1..6 over the field of char p

Note: activate this ring with the `setring` command
the variables are ordered as x(1),...x(6),y(1),...,y(6),Ha,Hb.

Example:

```
LIB "ncalg.lib";
def a = makeUg2();
a;
↦ // coefficients: QQ
↦ // number of vars : 14
↦ //        block   1 : ordering dp
↦ //                  : names    x(1) x(2) x(3) x(4) x(5) x(6) y(1) y(2) y(\
   3) y(4) y(5) y(6) Ha Hb
↦ //        block   2 : ordering C
↦ // noncommutative relations: ...
setring a;
// ...  56  noncommutative relations
```

See also: Section 7.7.12.3 [makeUgl], page 515; Section 7.7.12.2 [makeUsl], page 514.

7.7.12.18 makeUf4

Procedure from library `ncalg.lib` (see Section 7.7.12 [ncalg_lib], page 513).

Usage: makeUf4([p]); p an optional integer (field characteristic)

Return: a ring, describing U(f_4)

Note: You have to activate this ring with the 'setring' command. The presentation of U(f_4) is derived from the Chevalley representation of f_4, positive resp. negative roots are denoted by x(i) resp. y(i); Cartan elements are denoted by h(i).

Example:

```
LIB "ncalg.lib";
def ncAlgebra = makeUf4();
ncAlgebra;
↦ // coefficients: QQ
↦ // number of vars : 52
↦ //        block   1 : ordering dp
↦ //                  : names    X(1) X(2) X(3) X(4) X(5) X(6) X(7) X(8) X(\
   9) X(10) X(11) X(12) X(13) X(14) X(15) X(16) X(17) X(18) X(19) X(20) X(21\
   ) X(22) X(23) X(24) Y(1) Y(2) Y(3) Y(4) Y(5) Y(6) Y(7) Y(8) Y(9) Y(10) Y(\
   11) Y(12) Y(13) Y(14) Y(15) Y(16) Y(17) Y(18) Y(19) Y(20) Y(21) Y(22) Y(2\
   3) Y(24) H(1) H(2) H(3) H(4)
↦ //        block   2 : ordering C
↦ // noncommutative relations: ...
setring ncAlgebra;
// ...  552  noncommutative relations
```

See also: Section 7.7.12.19 [makeUe6], page 524; Section 7.7.12.20 [makeUe7], page 525; Section 7.7.12.21 [makeUe8], page 526; Section 7.7.12.17 [makeUg2], page 523; Section 7.7.12.2 [makeUsl], page 514; Section 7.7.12.4 [makeUso5], page 516; Section 7.7.12.12 [makeUsp1], page 520.

7.7.12.19 makeUe6

Procedure from library `ncalg.lib` (see Section 7.7.12 [ncalg_lib], page 513).

Usage: makeUe6([p]); p an optional integer (field characteristic)

Return: a ring, describing U(e_6)

Note: You have to activate this ring with the 'setring' command. The presentation of U(e_6) is derived from the Chevalley representation of e_6, positive resp. negative roots are denoted by x(i) resp. y(i); Cartan elements are denoted by h(i).

Example:

```
LIB "ncalg.lib";
def ncAlgebra = makeUe6();
ncAlgebra;
↦ // coefficients: QQ
↦ // number of vars : 78
↦ //        block   1 : ordering dp
↦ //                  : names    X(1) X(2) X(3) X(4) X(5) X(6) X(7) X(8) X(\
   9) X(10) X(11) X(12) X(13) X(14) X(15) X(16) X(17) X(18) X(19) X(20) X(21\
   ) X(22) X(23) X(24) X(25) X(26) X(27) X(28) X(29) X(30) X(31) X(32) X(33)\
    X(34) X(35) X(36) Y(1) Y(2) Y(3) Y(4) Y(5) Y(6) Y(7) Y(8) Y(9) Y(10) Y(1\
   1) Y(12) Y(13) Y(14) Y(15) Y(16) Y(17) Y(18) Y(19) Y(20) Y(21) Y(22) Y(23\
   ) Y(24) Y(25) Y(26) Y(27) Y(28) Y(29) Y(30) Y(31) Y(32) Y(33) Y(34) Y(35)\
    Y(36) H(1) H(2) H(3) H(4) H(5) H(6)
↦ //        block   2 : ordering C
↦ // noncommutative relations: ...
setring ncAlgebra;
// ...  1008  noncommutative relations
```

See also: Section 7.7.12.19 [makeUe6], page 524; Section 7.7.12.20 [makeUe7], page 525; Section 7.7.12.21 [makeUe8], page 526; Section 7.7.12.18 [makeUf4], page 524; Section 7.7.12.17 [makeUg2], page 523; Section 7.7.12.2 [makeUsl], page 514; Section 7.7.12.4 [makeUso5], page 516; Section 7.7.12.12 [makeUsp1], page 520.

7.7.12.20 makeUe7

Procedure from library `ncalg.lib` (see Section 7.7.12 [ncalg_lib], page 513).

Usage: makeUe7([p]); p an optional integer (field characteristic)

Return: a ring, describing U(e_7)

Note: You have to activate this ring with the 'setring' command. The presentation of U(e_7) is derived from the Chevalley representation of e_7, positive resp. negative roots are denoted by x(i) resp. y(i); Cartan elements are denoted by h(i).

Example:

```
LIB "ncalg.lib";
def ncAlgebra = makeUe7();
ncAlgebra;
↦ // coefficients: QQ
↦ // number of vars : 133
↦ //        block   1 : ordering dp
↦ //                  : names    X(1) X(2) X(3) X(4) X(5) X(6) X(7) X(8) X(\
   9) X(10) X(11) X(12) X(13) X(14) X(15) X(16) X(17) X(18) X(19) X(20) X(21\
   ) X(22) X(23) X(24) X(25) X(26) X(27) X(28) X(29) X(30) X(31) X(32) X(33)\
    X(34) X(35) X(36) X(37) X(38) X(39) X(40) X(41) X(42) X(43) X(44) X(45) \
   X(46) X(47) X(48) X(49) X(50) X(51) X(52) X(53) X(54) X(55) X(56) X(57) X\
   (58) X(59) X(60) X(61) X(62) X(63) Y(1) Y(2) Y(3) Y(4) Y(5) Y(6) Y(7) Y(8\
   ) Y(9) Y(10) Y(11) Y(12) Y(13) Y(14) Y(15) Y(16) Y(17) Y(18) Y(19) Y(20) \
```

```
   Y(21) Y(22) Y(23) Y(24) Y(25) Y(26) Y(27) Y(28) Y(29) Y(30) Y(31) Y(32) Y\
   (33) Y(34) Y(35) Y(36) Y(37) Y(38) Y(39) Y(40) Y(41) Y(42) Y(43) Y(44) Y(\
   45) Y(46) Y(47) Y(48) Y(49) Y(50) Y(51) Y(52) Y(53) Y(54) Y(55) Y(56) Y(5\
   7) Y(58) Y(59) Y(60) Y(61) Y(62) Y(63) H(1) H(2) H(3) H(4) H(5) H(6) H(7)
↦ //        block   2 : ordering C
↦ // noncommutative relations: ...
setring ncAlgebra;
// ...  2541  noncommutative relations
```

See also: Section 7.7.12.19 [makeUe6], page 524; Section 7.7.12.20 [makeUe7], page 525; Section 7.7.12.21 [makeUe8], page 526; Section 7.7.12.18 [makeUf4], page 524; Section 7.7.12.17 [makeUg2], page 523; Section 7.7.12.2 [makeUsl], page 514; Section 7.7.12.4 [makeUso5], page 516; Section 7.7.12.12 [makeUsp1], page 520.

7.7.12.21 makeUe8

Procedure from library `ncalg.lib` (see Section 7.7.12 [ncalg_lib], page 513).

Usage: makeUe8([p]); p an optional integer (field characteristic)

Return: a ring, describing U(e_8)

Note: You have to activate this ring with the 'setring' command. The presentation of U(e_8) is derived from the Chevalley representation of e_8, positive resp. negative roots are denoted by x(i) resp. y(i); Cartan elements are denoted by h(i).

Example:

```
LIB "ncalg.lib";
def ncAlgebra = makeUe8();
ncAlgebra;
↦ // coefficients: QQ
↦ // number of vars : 248
↦ //        block   1 : ordering dp
↦ //                  : names    X(1) X(2) X(3) X(4) X(5) X(6) X(7) X(8) X(\
   9) X(10) X(11) X(12) X(13) X(14) X(15) X(16) X(17) X(18) X(19) X(20) X(21\
   ) X(22) X(23) X(24) X(25) X(26) X(27) X(28) X(29) X(30) X(31) X(32) X(33)\
    X(34) X(35) X(36) X(37) X(38) X(39) X(40) X(41) X(42) X(43) X(44) X(45) \
   X(46) X(47) X(48) X(49) X(50) X(51) X(52) X(53) X(54) X(55) X(56) X(57) X\
   (58) X(59) X(60) X(61) X(62) X(63) X(64) X(65) X(66) X(67) X(68) X(69) X(\
   70) X(71) X(72) X(73) X(74) X(75) X(76) X(77) X(78) X(79) X(80) X(81) X(8\
   2) X(83) X(84) X(85) X(86) X(87) X(88) X(89) X(90) X(91) X(92) X(93) X(94\
   ) X(95) X(96) X(97) X(98) X(99) X(100) X(101) X(102) X(103) X(104) X(105)\
    X(106) X(107) X(108) X(109) X(110) X(111) X(112) X(113) X(114) X(115) X(\
   116) X(117) X(118) X(119) X(120) Y(1) Y(2) Y(3) Y(4) Y(5) Y(6) Y(7) Y(8) \
   Y(9) Y(10) Y(11) Y(12) Y(13) Y(14) Y(15) Y(16) Y(17) Y(18) Y(19) Y(20) Y(\
   21) Y(22) Y(23) Y(24) Y(25) Y(26) Y(27) Y(28) Y(29) Y(30) Y(31) Y(32) Y(3\
   3) Y(34) Y(35) Y(36) Y(37) Y(38) Y(39) Y(40) Y(41) Y(42) Y(43) Y(44) Y(45\
   ) Y(46) Y(47) Y(48) Y(49) Y(50) Y(51) Y(52) Y(53) Y(54) Y(55) Y(56) Y(57)\
    Y(58) Y(59) Y(60) Y(61) Y(62) Y(63) Y(64) Y(65) Y(66) Y(67) Y(68) Y(69) \
   Y(70) Y(71) Y(72) Y(73) Y(74) Y(75) Y(76) Y(77) Y(78) Y(79) Y(80) Y(81) Y\
   (82) Y(83) Y(84) Y(85) Y(86) Y(87) Y(88) Y(89) Y(90) Y(91) Y(92) Y(93) Y(\
   94) Y(95) Y(96) Y(97) Y(98) Y(99) Y(100) Y(101) Y(102) Y(103) Y(104) Y(10\
   5) Y(106) Y(107) Y(108) Y(109) Y(110) Y(111) Y(112) Y(113) Y(114) Y(115) \
   Y(116) Y(117) Y(118) Y(119) Y(120) H(1) H(2) H(3) H(4) H(5) H(6) H(7) H(8\
   )
```

```
↦ //        block   2 : ordering C
↦ // noncommutative relations: ...
setring ncAlgebra;
// ...  7752  noncommutative relations
```

See also: Section 7.7.12.19 [makeUe6], page 524; Section 7.7.12.20 [makeUe7], page 525; Section 7.7.12.21 [makeUe8], page 526; Section 7.7.12.18 [makeUf4], page 524; Section 7.7.12.17 [makeUg2], page 523; Section 7.7.12.2 [makeUsl], page 514; Section 7.7.12.4 [makeUso5], page 516; Section 7.7.12.12 [makeUsp1], page 520.

7.7.12.22 makeQso3

Procedure from library `ncalg.lib` (see Section 7.7.12 [ncalg_lib], page 513).

Usage: makeQso3([n]), n an optional int

Purpose: set up the U_q(so_3) in the presentation of Klimyk; if n is specified, the quantum parameter Q will be specialized at the (2n)-th root of unity

Return: ring

Note: activate this ring with the `setring` command

Example:

```
LIB "ncalg.lib";
def K = makeQso3(3);
setring K;
K;
↦ // coefficients: QQ[Q]/(Q2-Q+1)
↦ // number of vars : 3
↦ //        block   1 : ordering dp
↦ //                  : names    x y z
↦ //        block   2 : ordering C
↦ // noncommutative relations:
↦ //    yx=(Q-1)*xy+(-Q)*z
↦ //    zx=(-Q)*xz+(-Q+1)*y
↦ //    zy=(Q-1)*yz+(-Q)*x
```

See also: Section 7.7.12.25 [Qso3Casimir], page 529; Section 7.7.12.23 [makeQsl2], page 527; Section 7.7.12.24 [makeQsl3], page 528; Section 7.7.12.17 [makeUg2], page 523; Section 7.7.12.3 [makeUgl], page 515; Section 7.7.12.2 [makeUsl], page 514.

7.7.12.23 makeQsl2

Procedure from library `ncalg.lib` (see Section 7.7.12 [ncalg_lib], page 513).

Usage: makeQsl2([n]), n an optional int

Return: ring

Purpose: define the U_q(sl_2) as a factor-ring of a ring V_q(sl_2) modulo the ideal `Qideal`

Note: the output consists of a ring, presenting V_q(sl_2) together with the ideal called `Qideal` in this ring
activate this ring with the `setring` command
in order to create the U_q(sl_2) from the output, execute the command like `qring Usl2q = Qideal;`
If n is specified, the quantum parameter q will be specialized at the n-th root of unity

Example:

```
LIB "ncalg.lib";
def A = makeQsl2(3);
setring A;
Qideal;
↦ Qideal[1]=Ke*Kf-1
qring Usl2q = Qideal;
Usl2q;
↦ // coefficients: QQ[q]/(q^2+q+1)
↦ // number of vars : 4
↦ //        block   1 : ordering dp
↦ //                  : names    E F Ke Kf
↦ //        block   2 : ordering C
↦ // noncommutative relations:
↦ //    FE=E*F+(2/3*q+1/3)*Ke+(-2/3*q-1/3)*Kf
↦ //    KeE=(-q-1)*E*Ke
↦ //    KfE=(q)*E*Kf
↦ //    KeF=(q)*F*Ke
↦ //    KfF=(-q-1)*F*Kf
↦ // quotient ring from ideal
↦ _[1]=Ke*Kf-1
```

See also: Section 7.7.12.24 [makeQsl3], page 528; Section 7.7.12.22 [makeQso3], page 527; Section 7.7.12.2 [makeUsl], page 514.

7.7.12.24 makeQsl3

Procedure from library `ncalg.lib` (see Section 7.7.12 [ncalg_lib], page 513).

Usage: makeQsl3([n]), n an optional int

Return: ring

Purpose: define the U_q(sl_3) as a factor-ring of a ring V_q(sl_3) modulo the ideal `Qideal`

Note: the output consists of a ring, presenting V_q(sl_3) together with the ideal called `Qideal` in this ring
activate this ring with the `setring` command
in order to create the U_q(sl_3) from the output, execute the command like `qring Usl3q = Qideal;`
If n is specified, the quantum parameter q will be specialized at the n-th root of unity

Example:

```
LIB "ncalg.lib";
def B = makeQsl3(5);
setring B;
qring Usl3q = Qideal;
Usl3q;
↦ // coefficients: QQ[q]/(q^4+q^3+q^2+q+1)
↦ // number of vars : 10
↦ //        block   1 : ordering wp
↦ //                  : names    f12 f13 f23 k1 k2 l1 l2 e12 e13 e23
↦ //                  : weights    2   3   2   1   1   1   1   2   3   2
↦ //        block   2 : ordering C
↦ // noncommutative relations:
```

```
↦ //    f13f12=(q^3)*f12*f13
↦ //    f23f12=(q^2)*f12*f23+(-q)*f13
↦ //    k1f12=(q^3)*f12*k1
↦ //    k2f12=(q)*f12*k2
↦ //    l1f12=(q^2)*f12*l1
↦ //    l2f12=(-q^3-q^2-q-1)*f12*l2
↦ //    e12f12=f12*e12+(1/5*q^3-3/5*q^2-2/5*q-1/5)*k1^2+(-1/5*q^3+3/5*q^2+2\
   /5*q+1/5)*l1^2
↦ //    e13f12=f12*e13+(q^3+q^2+q+1)*l1^2*e23
↦ //    f23f13=(q^3)*f13*f23
↦ //    k1f13=(-q^3-q^2-q-1)*f13*k1
↦ //    k2f13=(-q^3-q^2-q-1)*f13*k2
↦ //    l1f13=(q)*f13*l1
↦ //    l2f13=(q)*f13*l2
↦ //    e12f13=f13*e12+(q)*f23*k1^2
↦ //    e13f13=f13*e13+(-1/5*q^3+3/5*q^2+2/5*q+1/5)*k1^2*k2^2+(1/5*q^3-3/5*\
   q^2-2/5*q-1/5)*l1^2*l2^2
↦ //    e23f13=f13*e23+(q^3+q^2+q+1)*f12*l2^2
↦ //    k1f23=(q)*f23*k1
↦ //    k2f23=(q^3)*f23*k2
↦ //    l1f23=(-q^3-q^2-q-1)*f23*l1
↦ //    l2f23=(q^2)*f23*l2
↦ //    e13f23=f23*e13+(q)*k2^2*e12
↦ //    e23f23=f23*e23+(1/5*q^3-3/5*q^2-2/5*q-1/5)*k2^2+(-1/5*q^3+3/5*q^2+2\
   /5*q+1/5)*l2^2
↦ //    e12k1=(q^3)*k1*e12
↦ //    e13k1=(-q^3-q^2-q-1)*k1*e13
↦ //    e23k1=(q)*k1*e23
↦ //    e12k2=(q)*k2*e12
↦ //    e13k2=(-q^3-q^2-q-1)*k2*e13
↦ //    e23k2=(q^3)*k2*e23
↦ //    e12l1=(q^2)*l1*e12
↦ //    e13l1=(q)*l1*e13
↦ //    e23l1=(-q^3-q^2-q-1)*l1*e23
↦ //    e12l2=(-q^3-q^2-q-1)*l2*e12
↦ //    e13l2=(q)*l2*e13
↦ //    e23l2=(q^2)*l2*e23
↦ //    e13e12=(q^3)*e12*e13
↦ //    e23e12=(q^2)*e12*e23+(-q)*e13
↦ //    e23e13=(q^3)*e13*e23
↦ // quotient ring from ideal
↦ _[1]=k2*l2-1
↦ _[2]=k1*l1-1
```

See also: Section 7.7.12.23 [makeQsl2], page 527; Section 7.7.12.22 [makeQso3], page 527; Section 7.7.12.2 [makeUsl], page 514.

7.7.12.25 Qso3Casimir

Procedure from library `ncalg.lib` (see Section 7.7.12 [ncalg_lib], page 513).

Usage: Qso3Casimir(n [,m]), n an integer, m an optional integer

Return: list (of polynomials)

Purpose: compute the Casimir (central) elements of U_q(so_3) for the quantum parameter specialized at the n-th root of unity; if m!=0 is given, polynomials will be normalized

Assume: the basering must be U_q(so_3)

Example:

```
LIB "ncalg.lib";
def R = makeQso3(5);
setring R;
list C = Qso3Casimir(5);
C;
↦ [1]:
↦    1/5*x5+(1/5Q3-1/5Q2+2/5)*x3+(1/5Q3-1/5Q2+1/5)*x
↦ [2]:
↦    1/5*y5+(1/5Q3-1/5Q2+2/5)*y3+(1/5Q3-1/5Q2+1/5)*y
↦ [3]:
↦    1/5*z5+(1/5Q3-1/5Q2+2/5)*z3+(1/5Q3-1/5Q2+1/5)*z
list Cnorm = Qso3Casimir(5,1);
Cnorm;
↦ [1]:
↦    x5+(Q3-Q2+2)*x3+(Q3-Q2+1)*x
↦ [2]:
↦    y5+(Q3-Q2+2)*y3+(Q3-Q2+1)*y
↦ [3]:
↦    z5+(Q3-Q2+2)*z3+(Q3-Q2+1)*z
```

See also: Section 7.7.12.22 [makeQso3], page 527.

7.7.12.26 GKZsystem

Procedure from library `ncalg.lib` (see Section 7.7.12 [ncalg_lib], page 513).

Usage: GKZsystem(A, sord, alg, [,v]); A intmat, sord, alg string, v intvec

Return: ring

Purpose: define a ring (Weyl algebra) and create a Gelfand-Kapranov-Zelevinsky (GKZ) system of equations in a ring from the following data:
`A` is an intmat, defining the system,
`sord` is a string with desired term ordering,
`alg` is a string, saying which algorithm to use (exactly like in toric_lib),
`v` is an optional intvec.
In addition, the ideal called `GKZid` containing actual equations is calculated and exported to the ring.

Note: activate the output ring with the `setring` command. This procedure is elaborated by Oleksandr Iena

Assume: This procedure uses toric_lib and therefore inherits its input requirements:
possible values for input variable `alg` are: "ect","pt","blr", "hs", "du".
As for the term ordering, it should be a string `sord` in Singular format like "lp","dp", etc.
Please consult the toric_lib for allowed orderings and more details.

Example:

```
LIB "ncalg.lib";
// example 3.1.4 from the [SST] without the vector w
intmat A[2][4]=3,2,1,0,0,1,2,3;
print(A);
↦      3     2     1     0
↦      0     1     2     3
def D1 = GKZsystem(A,"lp","ect");
setring D1;
D1;
↦ // coefficients: QQ(b(1), b(2))
↦ // number of vars : 8
↦ //        block   1 : ordering a
↦ //                  : names    x(1) x(2) x(3) x(4)
↦ //                  : weights     0    0    0    0
↦ //        block   2 : ordering lp
↦ //                  : names    x(1) x(2) x(3) x(4) d(1) d(2) d(3) d(4)
↦ //        block   3 : ordering C
↦ // noncommutative relations:
↦ //    d(1)x(1)=x(1)*d(1)+1
↦ //    d(2)x(2)=x(2)*d(2)+1
↦ //    d(3)x(3)=x(3)*d(3)+1
↦ //    d(4)x(4)=x(4)*d(4)+1
print(GKZid);
↦ 3*x(1)*d(1)+2*x(2)*d(2)+x(3)*d(3)+(-b(1)),
↦ x(2)*d(2)+2*x(3)*d(3)+3*x(4)*d(4)+(-b(2)),
↦ d(2)*d(4)-d(3)^2,
↦ d(1)*d(4)-d(2)*d(3),
↦ d(1)*d(3)-d(2)^2
// now, consider A with the vector w=1,1,1,1
intvec v=1,1,1,1;
def D2 = GKZsystem(A,"lp","blr",v);
setring D2;
print(GKZid);
↦ 3*x(1)*d(1)+2*x(2)*d(2)+x(3)*d(3)+(-b(1)),
↦ x(2)*d(2)+2*x(3)*d(3)+3*x(4)*d(4)+(-b(2)),
↦ d(2)*d(4)-d(3)^2,
↦ d(1)*d(4)-d(2)*d(3),
↦ d(1)*d(3)-d(2)^2
```

See also: Section D.4.29 [toric_lib], page 762.

7.7.13 ncdecomp_lib

Library: ncdecomp.lib

Purpose: Decomposition of a module into its central characters

Authors: Viktor Levandovskyy, levandov@mathematik.uni-kl.de.

Overview:

This library presents algorithms for the central character decomposition of a module, i.e. a decomposition into generalized weight modules with respect to the center. Based on ideas of O. Khomenko and V. Levandovskyy (see the article [L2] in the References for details).

Procedures:

7.7.13.1 CentralQuot

Procedure from library `ncdecomp.lib` (see Section 7.7.13 [ncdecomp_lib], page 531).

Usage: CentralQuot(M, G), M a module, G an ideal

Assume: G is an ideal in the center of the base ring

Return: module

Purpose: compute the central quotient M:G

Theory: for an ideal G of the center of an algebra and a submodule M of A^n,
the central quotient of M by G is defined to be
M:G := { v in A^n | z*v in M, for all z in G }.

Note: the output module is not necessarily given in a Groebner basis

Example:

```
LIB "ncdecomp.lib";
option(returnSB);
def a = makeUsl2();
setring a;
ideal I = e3,f3,h3-4*h;
I = std(I);
poly C=4*e*f+h^2-2*h;  // C in Z(U(sl2)), the central element
ideal G = (C-8)*(C-24);  // G normal factor in Z(U(sl2)) as an ideal in the center
ideal R = CentralQuot(I,G);  // same as I:G
R;
↦ R[1]=h
↦ R[2]=f
↦ R[3]=e
```

See also: Section 7.7.13.3 [CenCharDec], page 533; Section 7.7.13.2 [CentralSaturation], page 532.

7.7.13.2 CentralSaturation

Procedure from library `ncdecomp.lib` (see Section 7.7.13 [ncdecomp_lib], page 531).

Usage: CentralSaturation(M, T), for a module M and an ideal T

Assume: T is an ideal in the center of the base ring

Return: module

Purpose: compute the central saturation of M by T, that is M:T^{\infty}, by repititive application of `CentralQuot`

Note: the output module is not necessarily a Groebner basis

Example:

```
LIB "ncdecomp.lib";
option(returnSB);
def a = makeUsl2();
setring a;
ideal I = e3,f3,h3-4*h;
I = std(I);
poly C=4*e*f+h^2-2*h;
ideal G = C*(C-8);
ideal R = CentralSaturation(I,G);
```

```
R=std(R);
vdim(R);
↦ 5
R;
↦ R[1]=h
↦ R[2]=ef-6
↦ R[3]=f3
↦ R[4]=e3
```

See also: Section 7.7.13.3 [CenCharDec], page 533; Section 7.7.13.1 [CentralQuot], page 532.

7.7.13.3 CenCharDec

Procedure from library `ncdecomp.lib` (see Section 7.7.13 [ncdecomp_lib], page 531).

Usage: CenCharDec(I, C); I a module, C an ideal

Assume: C consists of generators of the center of the base ring

Return: a list L, where each entry consists of three records (if a finite decomposition exists)
L[*][1] ('ideal' type), the central character as a maximal ideal in the center,
L[*][2] ('module' type), the Groebner basis of the weight module, corresponding to the character in L[*][1],
L[*][3] ('int' type) is the vector space dimension of the weight module (-1 in case of infinite dimension);

Purpose: compute a finite decomposition of C into central characters or determine that there is no finite decomposition

Note: actual decomposition is the sum of L[i][2] above;
some modules have no finite decomposition (in such case one gets warning message)
The function `central` in `central.lib` may be used to obtain C, when needed.

Example:

```
LIB "ncdecomp.lib";
printlevel=0;
option(returnSB);
def a = makeUsl2(); // U(sl_2) in characteristic 0
setring a;
ideal I = e3,f3,h3-4*h;
I = twostd(I);           // two-sided ideal generated by I
vdim(I);                 // it is finite-dimensional
↦ 10
ideal Cn = 4*e*f+h^2-2*h; // the only central element
list T = CenCharDec(I,Cn);
T;
↦ [1]:
↦    [1]:
↦       _[1]=4ef+h2-2h-8
↦    [2]:
↦       _[1]=h
↦       _[2]=f
↦       _[3]=e
↦    [3]:
↦       1
↦ [2]:
```

```
↦    [1]:
↦       _[1]=4ef+h2-2h
↦    [2]:
↦       _[1]=4ef+h2-2h-8
↦       _[2]=h3-4h
↦       _[3]=fh2-2fh
↦       _[4]=eh2+2eh
↦       _[5]=f2h-2f2
↦       _[6]=e2h+2e2
↦       _[7]=f3
↦       _[8]=e3
↦    [3]:
↦       9
// consider another example
ideal J = e*f*h;
CenCharDec(J,Cn);
↦ There is no finite decomposition
↦ 0
```

See also: Section 7.7.13.1 [CentralQuot], page 532; Section 7.7.13.2 [CentralSaturation], page 532.

7.7.13.4 IntersectWithSub

Procedure from library `ncdecomp.lib` (see Section 7.7.13 [ncdecomp_lib], page 531).

Usage: IntersectWithSub(M,Z), M an ideal, Z an ideal

Assume: Z consists of pairwise commutative elements

Return: ideal of two-sided generators, not a Groebner basis

Purpose: computes the intersection of M with the subalgebra, generated by Z

Note: usually Z consists of generators of the center
The function `central` from `central.lib` may be used to obtain the center Z, if needed.

Example:

```
LIB "ncdecomp.lib";
ring R=(0,a),(e,f,h),Dp;
matrix @d[3][3];
@d[1,2]=-h;   @d[1,3]=2e;   @d[2,3]=-2f;
def r = nc_algebra(1,@d); setring r; // parametric U(sl_2)
ideal I = e,h-a;
ideal C;
C[1] = h^2-2*h+4*e*f; // the center of U(sl_2)
ideal X = IntersectWithSub(I,C);
X;
↦ X[1]=4*ef+h2-2*h+(-a2-2a)
ideal G = e*f, h; // the biggest comm. subalgebra of U(sl_2)
ideal Y = IntersectWithSub(I,G);
Y;
↦ Y[1]=h+(-a)
↦ Y[2]=ef+(-a)
```

7.7.14 ncfactor_lib

Library: ncfactor.lib

Purpose: Tools for factorization in some noncommutative algebras

Authors: Albert Heinle, aheinle@uwaterloo.ca
Viktor Levandovskyy, levandov@math.rwth-aachen.de

Overview: New methods for factorization on polynomials
are implemented for three types of algebras, all generated by 2*n, n in NN, generators (n'th Weyl, n'th shift and graded polynomials in n'th q-Weyl algebras) over a field K. More detailled description of the algorithms and related publications can be found at @url{https://cs.uwaterloo.ca/~aheinle/}.

Procedures:

7.7.14.1 ncfactor

Procedure from library `ncfactor.lib` (see Section 7.7.14 [ncfactor_lib], page 534).

Usage: ncfactor(h); h is a polynomial in a non-commutative polynomial algebra over a field k.

Return: list(list)

Purpose: Compute all factorizations of h.

Theory: Implements an ansatz-driven factorization method as outlined by Bell, Heinle and Levandovskyy in "On Noncommutative Finite Factorization Domains".

Assume: - k is a ring, such that factorize can factor any univariate and multivariate commutative polynomial over k.
- There exists at least one variable in the ring.

Note: - Every entry of the output list is a list with factors for one possible factorization. The first factor is always a constant (1, if no nontrivial constant could be excluded).

Example:

```
LIB "ncfactor.lib";
def R = makeUsl2();
setring(R);
poly p = e^3*f+e^2*f^2-e^3+e^2*f+2*e*f^2-3*e^2*h-2*e*f*h-8*e^2
+e*f+f^2-4*e*h-2*f*h-7*e+f-h;
ncfactor(p);
↦ [1]:
↦    [1]:
↦ 1
↦    [2]:
↦       e+1
↦    [3]:
↦       ef-e+f-2h-3
↦    [4]:
↦       e+f
↦ [2]:
↦    [1]:
↦ 1
↦    [2]:
↦       e2f+ef2-e2+f2-2eh-3e-f-2h
↦    [3]:
↦       e+1
```

See also: Section 7.7.14.5 [facSubWeyl], page 538; Section 7.7.14.2 [facWeyl], page 536; Section 7.7.14.4 [testNCfac], page 537.

7.7.14.2 facWeyl

Procedure from library `ncfactor.lib` (see Section 7.7.14 [ncfactor_lib], page 534).

Usage: facWeyl(h); h a polynomial in the nth Weyl algebra

Return: list

Purpose: compute all factorizations of a polynomial in the first Weyl algebra

Theory: Implements the new algorithm by A. Heinle and V. Levandovskyy, see the thesis of A. Heinle

Assume: basering is the nth Weyl algebra, where n in NN.

Note: Every entry of the output list is a list with factors for one possible factorization. The first factor is always a constant (1, if no nontrivial constant could be excluded).

Example:

```
LIB "ncfactor.lib";
ring R = 0,(x1,x2,d1,d2),dp;
matrix C[4][4] = 1,1,1,1,
1,1,1,1,
1,1,1,1,
1,1,1,1;
matrix D[4][4] = 0,0,1,0,
0,0,0,1,
-1,0,0,0,
0,-1,0,0;
def r = nc_algebra(C,D);
setring(r);
poly h = (d1+1)^2*(d1 + x1*d2);
facWeyl(h);
↦ [1]:
↦    [1]:
↦ 1
↦    [2]:
↦       d1+1
↦    [3]:
↦       d1+1
↦    [4]:
↦       x1*d2+d1
↦ [2]:
↦    [1]:
↦ 1
↦    [2]:
↦       x1*d1*d2+d1^2+x1*d2+d1+2*d2
↦    [3]:
↦       d1+1
```

See also: Section 7.7.14.7 [facFirstShift], page 540; Section 7.7.14.3 [facFirstWeyl], page 537; Section 7.7.14.5 [facSubWeyl], page 538; Section 7.7.14.4 [testNCfac], page 537.

7.7.14.3 facFirstWeyl

Procedure from library `ncfactor.lib` (see Section 7.7.14 [ncfactor_lib], page 534).

Usage: facFirstWeyl(h); h a polynomial in the first Weyl algebra

Return: list

Purpose: compute all factorizations of a polynomial in the first Weyl algebra

Theory: This function is a wrapper for facWeyl. It exists to make this library downward-compatible with older versions.

Assume: basering is the first Weyl algebra

Note: Every entry of the output list is a list with factors for one possible factorization. The first factor is always a constant (1, if no nontrivial constant could be excluded).

Example:

```
LIB "ncfactor.lib";
ring R = 0,(x,y),dp;
def r = nc_algebra(1,1);
setring(r);
poly h = (x^2*y^2+x)*(x+1);
facFirstWeyl(h);
↦ [1]:
↦    [1]:
↦ 1
↦    [2]:
↦       x
↦    [3]:
↦       xy2+1
↦    [4]:
↦       x+1
```

See also: Section 7.7.14.6 [facShift], page 539; Section 7.7.14.5 [facSubWeyl], page 538; Section 7.7.14.4 [testNCfac], page 537.

7.7.14.4 testNCfac

Procedure from library `ncfactor.lib` (see Section 7.7.14 [ncfactor_lib], page 534).

Usage: testNCfac(l[,p,b]); l is a list, p is an optional poly, b is 1 or 0

Return: Case 1: No optional argument. In this case the output is 1, if the entries in the given list represent the same polynomial or 0 otherwise.
Case 2: One optional argument p is given. In this case it returns 1, if all the entries in l are factorizations of p, otherwise 0. Case 3: Second optional b is given. In this case a list is returned containing the difference between the product of each entry in l and p.

Assume: basering is the first Weyl algebra, the entries of l are polynomials

Purpose: Checks whether a list of factorizations contains factorizations of the same element in the first Weyl algebra

Theory: `testNCfac` multiplies out each factorization and checks whether each factorization was a factorization of the same element.
- if there is only a list given, the output will be 0, if it does not contain factorizations of the same element. Otherwise the output will be 1.

- if there is a polynomial in the second argument, then the procedure checks whether the given list contains factorizations of this polynomial. If it does, then the output depends on the third argument. If it is not given, the procedure will check whether the factorizations in the list l are associated to this polynomial and return either 1 or 0, respectively. If the third argument is given, the output will be a list with the length of the given one and in each entry is the product of one entry in l subtracted by the polynomial.

Example:

```
LIB "ncfactor.lib";
ring r = 0,(x,y),dp;
def R = nc_algebra(1,1);
setring R;
poly h = (x^2*y^2+1)*(x^2);
def t1 = facFirstWeyl(h);
//fist a correct list
testNCfac(t1);
↦ 1
//now a correct list with the factorized polynomial
testNCfac(t1,h);
↦ 1
//now we put in an incorrect list without a polynomial
t1[3][3] = y;
testNCfac(t1);
↦ 0
// take h as additional input
testNCfac(t1,h);
↦ 0
// take h as additional input and output list of differences
testNCfac(t1,h,1);
↦ [1]:
↦    0
↦ [2]:
↦    0
↦ [3]:
↦    -x4y2+x3y3-4x3y+3x2y2-3x2+xy+1
```

See also: Section 7.7.14.7 [facFirstShift], page 540; Section 7.7.14.3 [facFirstWeyl], page 537; Section 7.7.14.5 [facSubWeyl], page 538.

7.7.14.5 facSubWeyl

Procedure from library `ncfactor.lib` (see Section 7.7.14 [ncfactor_lib], page 534).

Usage: facSubWeyl(h,[x_1,...,x_n,d_1,...,d_n]); h is a polynomial, x_i, d_i are variables in the basering for i in {1,...,n}

Return: list(list)

Assume: x_i, d_i are variables of a basering with d_i*x_i = x_i*d_i+1 for i in {1,...,n}.
That is, they generate the copy of the first Weyl algebra in a basering.
Moreover, h is a polynomial in the x_i,d_i only.
If the list of variables is ommitted, this function will try to figure out itself if h is in a subalgebra that resembles the Weyl algebra.
This function produces an error if the conditions on the variables do not line up or if the

variables contained in h do not belong to a subalgebra of the basering that resembles the Weyl algebra.

Purpose: compute factorizations of the polynomial, depending on x_i and d_i.

Example:

```
LIB "ncfactor.lib";
ring r = 0,(x,y,z),dp;
matrix D[3][3]; D[1,3]=-1;
def R = nc_algebra(1,D); // x,z generate Weyl subalgebra
setring R;
poly h = (x^2*z^2+x)*x;
list fact1 = facSubWeyl(h,x,z);
// compare with facFirstWeyl:
ring s = 0,(z,x),dp;
def S = nc_algebra(1,1); setring S;
poly h = (x^2*z^2+x)*x;
list fact2 = facFirstWeyl(h);
map F = R,x,0,z;
list fact1 = F(fact1); // it is identical to list fact2
testNCfac(fact1); // check the correctness again
↦ 1
```

See also: Section 7.7.14.7 [facFirstShift], page 540; Section 7.7.14.3 [facFirstWeyl], page 537; Section 7.7.14.4 [testNCfac], page 537.

7.7.14.6 facShift

Procedure from library `ncfactor.lib` (see Section 7.7.14 [ncfactor_lib], page 534).

Usage: facShift(h); h a polynomial in the n'th shift algebra

Return: list

Purpose: compute all factorizations of a polynomial in the nth shift algebra

Theory: Currently, we do not have a specialized algorithm for the shift algebra in this library that takes advantage of the graded structure, hence this function is mapping to the general factorization algorithm for G-Algebras

Note: Every entry of the output list is a list with factors for one possible factorization.

Example:

```
LIB "ncfactor.lib";
ring R = 0,(x1,x2,s1,s2),dp;
matrix C[4][4] = 1,1,1,1,
1,1,1,1,
1,1,1,1,
1,1,1,1;
matrix D[4][4] = 0,0,s1,0,
0,0,0,s2,
-s1,0,0,0,
0,-s2,0,0;
def r = nc_algebra(C,D);
setring(r);
poly h = x1*(x1+1)*s1^2-2*x1*(x1+100)*s1+(x1+99)*(x1+100);
facShift(h);
```

```
↦ [1]:
↦    [1]:
↦ 1
↦    [2]:
↦       x1*s1-x1+s1-100
↦    [3]:
↦       x1*s1-x1-s1-99
↦ [2]:
↦    [1]:
↦ 1
↦    [2]:
↦       x1*s1-x1-100
↦    [3]:
↦       x1*s1-x1-99
↦ [3]:
↦    [1]:
↦ 1
↦    [2]:
↦       x1*s1-x1-99
↦    [3]:
↦       x1*s1-x1-100
```

See also: Section 7.7.14.3 [facFirstWeyl], page 537; Section 7.7.14.5 [facSubWeyl], page 538; Section 7.7.14.4 [testNCfac], page 537.

7.7.14.7 facFirstShift

Procedure from library `ncfactor.lib` (see Section 7.7.14 [ncfactor_lib], page 534).

Usage: facFirstShift(h); h a polynomial in the first shift algebra

Return: list

Purpose: compute all factorizations of a polynomial in the first shift algebra

Theory: This function is a wrapper for facShift. It exists to make this library downward-compatible with older versions.

Assume: basering is the first shift algebra

Note: Every entry of the output list is a list with factors for one possible factorization.

Example:

```
LIB "ncfactor.lib";
ring R = 0,(x,s),dp;
def r = nc_algebra(1,s);
setring(r);
poly h = (s^2*x+x)*s;
facFirstShift(h);
↦ [1]:
↦    [1]:
↦ 1
↦    [2]:
↦       s
↦    [3]:
↦       s2+1
↦    [4]:
```

```
↦        x-1
↦ [2]:
↦    [1]:
↦ 1
↦    [2]:
↦       s2+1
↦    [3]:
↦       s
↦    [4]:
↦       x-1
↦ [3]:
↦    [1]:
↦ 1
↦    [2]:
↦       s2+1
↦    [3]:
↦       x
↦    [4]:
↦       s
```

See also: Section 7.7.14.3 [facFirstWeyl], page 537; Section 7.7.14.5 [facSubWeyl], page 538; Section 7.7.14.4 [testNCfac], page 537.

7.7.14.8 homogfacNthQWeyl

Procedure from library `ncfactor.lib` (see Section 7.7.14 [ncfactor_lib], page 534).

Usage: homogfacNthQWeyl(h); h is a homogeneous polynomial in the n'th q-Weyl algebra with respect to the weight vector
`@` [-1,...,-1,1,...,1].
`@` __ __/ __ __/ `@` \/ \/ `@` n/2 n/2

Return: list

Purpose: Computes a factorization of a homogeneous polynomial h in the n'th q-Weyl algebra

Theory: `homogfacNthQWeyl` returns a list with a factorization of the given, [-1,1]-homogeneous polynomial. For every i in 1..n: If the degree of the polynomial in [d_i,x_i] is k with k positive, the last entries in the output list are the second variable. If k is positive, the last k entries will be x_i. The other entries will be irreducible polynomials of degree zero or 1 resp. -1. resp. other variables

General assumptions:
- The basering is the nth Weyl algebra and has the form, that the first n variables represent x1, ..., xn, and the second n variables do represent the d1, ..., dn.
- We have n parameters q_1,..., q_n given.

Example:

```
LIB "ncfactor.lib";
ring R = (0,q1,q2,q3),(x1,x2,x3,d1,d2,d3),dp;
matrix C[6][6] = 1,1,1,q1,1,1,
1,1,1,1,q2,1,
1,1,1,1,1,q3,
1,1,1,1,1,1,
1,1,1,1,1,1,
1,1,1,1,1,1;
```

```
matrix D[6][6] = 0,0,0,1,0,0,
0,0,0,0,1,0,
0,0,0,0,0,1,
-1,0,0,0,0,0,
0,-1,0,0,0,0,
0,0,-1,0,0,0;
def r = nc_algebra(C,D);
setring(r);
poly h =x1*x2^2*x3^3*d1*d2^2+x2*x3^3*d2;
homogfacNthQWeyl(h);
↦ [1]:
↦    1/(q2)
↦ [2]:
↦    x1*x2*d1*d2-x1*d1+(q2)
↦ [3]:
↦    x2
↦ [4]:
↦    d2
↦ [5]:
↦    x3
↦ [6]:
↦    x3
↦ [7]:
↦    x3
```

See also: Section 7.7.14.9 [homogfacFirstQWeyl], page 542; Section 7.7.14.11 [homogfacFirstQWeyl_all], page 560; Section 7.7.14.10 [homogfacNthQWeyl_all], page 543.

7.7.14.9 homogfacFirstQWeyl

Procedure from library `ncfactor.lib` (see Section 7.7.14 [ncfactor_lib], page 534).

Usage: homogfacFirstQWeyl(h); h is a homogeneous polynomial in the first q-Weyl algebra with respect to the weight vector [-1,1]

Return: list

Purpose: Computes a factorization of a homogeneous polynomial h with respect to the weight vector [-1,1] in the first q-Weyl algebra

Theory: This function is a wrapper for homogfacNthQWeyl. It exists to make this library downward-compatible with older versions.

Example:

```
LIB "ncfactor.lib";
ring R = (0,q),(x,d),dp;
def r = nc_algebra (q,1);
setring(r);
poly h = q^25*x^10*d^10+q^16*(q^4+q^3+q^2+q+1)^2*x^9*d^9+
q^9*(q^13+3*q^12+7*q^11+13*q^10+20*q^9+26*q^8+30*q^7+
31*q^6+26*q^5+20*q^4+13*q^3+7*q^2+3*q+1)*x^8*d^8+
q^4*(q^9+2*q^8+4*q^7+6*q^6+7*q^5+8*q^4+6*q^3+
4*q^2+2q+1)*(q^4+q^3+q^2+q+1)*(q^2+q+1)*x^7*d^7+
q*(q^2+q+1)*(q^5+2*q^4+2*q^3+3*q^2+2*q+1)*(q^4+q^3+q^2+q+1)*(q^2+1)*(q+1)*x^6*d^6+
(q^10+5*q^9+12*q^8+21*q^7+29*q^6+33*q^5+31*q^4+24*q^3+15*q^2+7*q+12)*x^5*d^5+
6*x^3*d^3+24;
```

```
homogfacFirstQWeyl(h);
↦ [1]:
↦    1
↦ [2]:
↦    x5d5+x3d3+4
↦ [3]:
↦    x5d5+6
```

See also: Section 7.7.14.11 [homogfacFirstQWeyl_all], page 560.

7.7.14.10 homogfacNthQWeyl_all

Procedure from library `ncfactor.lib` (see Section 7.7.14 [ncfactor_lib], page 534).

Usage: homogfacNthQWeyl_all(h); h is a homogeneous polynomial in the n'th q-Weyl algebra with respect to the weight vector
`@` [-1,...,-1,1,...,1].
`@` __ __/ __ __/ `@` \\/ \\/ `@` n/2 n/2

Return: list

Purpose: Computes all factorizations of a homogeneous polynomial h in the n'th q-Weyl algebra

Theory: `homogfacNthQWeyl` returns a list with lists representing each a factorization of the given,
[-1,...,-1,1,...,1]-homogeneous polynomial.

General assumptions:
- The basering is the nth Weyl algebra and has the form, that the first n variables represent x1, ..., xn, and the second n variables do represent the d1, ..., dn. - We have n parameters q_1,..., q_n given.

Example:

```
LIB "ncfactor.lib";
ring R = (0,q1,q2,q3),(x1,x2,x3,d1,d2,d3),dp;
matrix C[6][6] = 1,1,1,q1,1,1,
1,1,1,1,q2,1,
1,1,1,1,1,q3,
1,1,1,1,1,1,
1,1,1,1,1,1,
1,1,1,1,1,1;
matrix D[6][6] = 0,0,0,1,0,0,
0,0,0,0,1,0,
0,0,0,0,0,1,
-1,0,0,0,0,0,
0,-1,0,0,0,0,
0,0,-1,0,0,0;
def r = nc_algebra(C,D);
setring(r);
poly h =x1*x2^2*x3^3*d1*d2^2+x2*x3^3*d2;
homogfacNthQWeyl_all(h);
↦ [1]:
↦    [1]:
↦ 1
↦    [2]:
↦       x2
```

```
↦    [3]:
↦       x1*x2*d1*d2+1
↦    [4]:
↦       d2
↦    [5]:
↦       x3
↦    [6]:
↦       x3
↦    [7]:
↦       x3
↦ [2]:
↦    [1]:
↦ 1
↦    [2]:
↦       x2
↦    [3]:
↦       x1*x2*d1*d2+1
↦    [4]:
↦       x3
↦    [5]:
↦       d2
↦    [6]:
↦       x3
↦    [7]:
↦       x3
↦ [3]:
↦    [1]:
↦ 1
↦    [2]:
↦       x2
↦    [3]:
↦       x1*x2*d1*d2+1
↦    [4]:
↦       x3
↦    [5]:
↦       x3
↦    [6]:
↦       d2
↦    [7]:
↦       x3
↦ [4]:
↦    [1]:
↦ 1
↦    [2]:
↦       x2
↦    [3]:
↦       x1*x2*d1*d2+1
↦    [4]:
↦       x3
↦    [5]:
↦       x3
↦    [6]:
↦       x3
```

```
↦    [7]:
↦       d2
↦ [5]:
↦    [1]:
↦ 1
↦    [2]:
↦       x2
↦    [3]:
↦       x3
↦    [4]:
↦       x1*x2*d1*d2+1
↦    [5]:
↦       d2
↦    [6]:
↦       x3
↦    [7]:
↦       x3
↦ [6]:
↦    [1]:
↦ 1
↦    [2]:
↦       x2
↦    [3]:
↦       x3
↦    [4]:
↦       x1*x2*d1*d2+1
↦    [5]:
↦       x3
↦    [6]:
↦       d2
↦    [7]:
↦       x3
↦ [7]:
↦    [1]:
↦ 1
↦    [2]:
↦       x2
↦    [3]:
↦       x3
↦    [4]:
↦       x1*x2*d1*d2+1
↦    [5]:
↦       x3
↦    [6]:
↦       x3
↦    [7]:
↦       d2
↦ [8]:
↦    [1]:
↦ 1
↦    [2]:
↦       x2
↦    [3]:
```

```
↦        x3
↦     [4]:
↦        x3
↦     [5]:
↦        x1*x2*d1*d2+1
↦     [6]:
↦        d2
↦     [7]:
↦        x3
↦  [9]:
↦     [1]:
↦  1
↦     [2]:
↦        x2
↦     [3]:
↦        x3
↦     [4]:
↦        x3
↦     [5]:
↦        x1*x2*d1*d2+1
↦     [6]:
↦        x3
↦     [7]:
↦        d2
↦  [10]:
↦     [1]:
↦  1
↦     [2]:
↦        x2
↦     [3]:
↦        x3
↦     [4]:
↦        x3
↦     [5]:
↦        x3
↦     [6]:
↦        x1*x2*d1*d2+1
↦     [7]:
↦        d2
↦  [11]:
↦     [1]:
↦  1
↦     [2]:
↦        x3
↦     [3]:
↦        x2
↦     [4]:
↦        x1*x2*d1*d2+1
↦     [5]:
↦        d2
↦     [6]:
↦        x3
↦     [7]:
```

```
↦        x3
↦ [12]:
↦    [1]:
↦ 1
↦    [2]:
↦       x3
↦    [3]:
↦       x2
↦    [4]:
↦       x1*x2*d1*d2+1
↦    [5]:
↦       x3
↦    [6]:
↦       d2
↦    [7]:
↦       x3
↦ [13]:
↦    [1]:
↦ 1
↦    [2]:
↦       x3
↦    [3]:
↦       x2
↦    [4]:
↦       x1*x2*d1*d2+1
↦    [5]:
↦       x3
↦    [6]:
↦       x3
↦    [7]:
↦       d2
↦ [14]:
↦    [1]:
↦ 1
↦    [2]:
↦       x3
↦    [3]:
↦       x2
↦    [4]:
↦       x3
↦    [5]:
↦       x1*x2*d1*d2+1
↦    [6]:
↦       d2
↦    [7]:
↦       x3
↦ [15]:
↦    [1]:
↦ 1
↦    [2]:
↦       x3
↦    [3]:
↦       x2
```

```
↦     [4]:
↦        x3
↦     [5]:
↦        x1*x2*d1*d2+1
↦     [6]:
↦        x3
↦     [7]:
↦        d2
↦  [16]:
↦     [1]:
↦  1
↦     [2]:
↦        x3
↦     [3]:
↦        x2
↦     [4]:
↦        x3
↦     [5]:
↦        x3
↦     [6]:
↦        x1*x2*d1*d2+1
↦     [7]:
↦        d2
↦  [17]:
↦     [1]:
↦  1
↦     [2]:
↦        x3
↦     [3]:
↦        x3
↦     [4]:
↦        x2
↦     [5]:
↦        x1*x2*d1*d2+1
↦     [6]:
↦        d2
↦     [7]:
↦        x3
↦  [18]:
↦     [1]:
↦  1
↦     [2]:
↦        x3
↦     [3]:
↦        x3
↦     [4]:
↦        x2
↦     [5]:
↦        x1*x2*d1*d2+1
↦     [6]:
↦        x3
↦     [7]:
↦        d2
```

```
↦ [19]:
↦    [1]:
↦ 1
↦    [2]:
↦       x3
↦    [3]:
↦       x3
↦    [4]:
↦       x2
↦    [5]:
↦       x3
↦    [6]:
↦       x1*x2*d1*d2+1
↦    [7]:
↦       d2
↦ [20]:
↦    [1]:
↦ 1
↦    [2]:
↦       x3
↦    [3]:
↦       x3
↦    [4]:
↦       x3
↦    [5]:
↦       x2
↦    [6]:
↦       x1*x2*d1*d2+1
↦    [7]:
↦       d2
↦ [21]:
↦    [1]:
↦ 1/(q2)
↦    [2]:
↦       x1*x2*d1*d2-x1*d1+(q2)
↦    [3]:
↦       x2
↦    [4]:
↦       d2
↦    [5]:
↦       x3
↦    [6]:
↦       x3
↦    [7]:
↦       x3
↦ [22]:
↦    [1]:
↦ 1/(q2)
↦    [2]:
↦       x1*x2*d1*d2-x1*d1+(q2)
↦    [3]:
↦       x2
↦    [4]:
```

```
↦        x3
↦     [5]:
↦        d2
↦     [6]:
↦        x3
↦     [7]:
↦        x3
↦ [23]:
↦     [1]:
↦ 1/(q2)
↦     [2]:
↦        x1*x2*d1*d2-x1*d1+(q2)
↦     [3]:
↦        x2
↦     [4]:
↦        x3
↦     [5]:
↦        x3
↦     [6]:
↦        d2
↦     [7]:
↦        x3
↦ [24]:
↦     [1]:
↦ 1/(q2)
↦     [2]:
↦        x1*x2*d1*d2-x1*d1+(q2)
↦     [3]:
↦        x2
↦     [4]:
↦        x3
↦     [5]:
↦        x3
↦     [6]:
↦        x3
↦     [7]:
↦        d2
↦ [25]:
↦     [1]:
↦ 1/(q2)
↦     [2]:
↦        x1*x2*d1*d2-x1*d1+(q2)
↦     [3]:
↦        x3
↦     [4]:
↦        x2
↦     [5]:
↦        d2
↦     [6]:
↦        x3
↦     [7]:
↦        x3
↦ [26]:
```

```
↦    [1]:
↦ 1/(q2)
↦    [2]:
↦       x1*x2*d1*d2-x1*d1+(q2)
↦    [3]:
↦       x3
↦    [4]:
↦       x2
↦    [5]:
↦       x3
↦    [6]:
↦       d2
↦    [7]:
↦       x3
↦ [27]:
↦    [1]:
↦ 1/(q2)
↦    [2]:
↦       x1*x2*d1*d2-x1*d1+(q2)
↦    [3]:
↦       x3
↦    [4]:
↦       x2
↦    [5]:
↦       x3
↦    [6]:
↦       x3
↦    [7]:
↦       d2
↦ [28]:
↦    [1]:
↦ 1/(q2)
↦    [2]:
↦       x1*x2*d1*d2-x1*d1+(q2)
↦    [3]:
↦       x3
↦    [4]:
↦       x3
↦    [5]:
↦       x2
↦    [6]:
↦       d2
↦    [7]:
↦       x3
↦ [29]:
↦    [1]:
↦ 1/(q2)
↦    [2]:
↦       x1*x2*d1*d2-x1*d1+(q2)
↦    [3]:
↦       x3
↦    [4]:
↦       x3
```

```
↦    [5]:
↦       x2
↦    [6]:
↦       x3
↦    [7]:
↦       d2
↦ [30]:
↦    [1]:
↦ 1/(q2)
↦    [2]:
↦       x1*x2*d1*d2-x1*d1+(q2)
↦    [3]:
↦       x3
↦    [4]:
↦       x3
↦    [5]:
↦       x3
↦    [6]:
↦       x2
↦    [7]:
↦       d2
↦ [31]:
↦    [1]:
↦ 1/(q2)
↦    [2]:
↦       x2
↦    [3]:
↦       d2
↦    [4]:
↦       x1*x2*d1*d2-x1*d1+(q2)
↦    [5]:
↦       x3
↦    [6]:
↦       x3
↦    [7]:
↦       x3
↦ [32]:
↦    [1]:
↦ 1/(q2)
↦    [2]:
↦       x2
↦    [3]:
↦       d2
↦    [4]:
↦       x3
↦    [5]:
↦       x1*x2*d1*d2-x1*d1+(q2)
↦    [6]:
↦       x3
↦    [7]:
↦       x3
↦ [33]:
↦    [1]:
```

```
↦ 1/(q2)
↦    [2]:
↦       x2
↦    [3]:
↦       d2
↦    [4]:
↦       x3
↦    [5]:
↦       x3
↦    [6]:
↦       x1*x2*d1*d2-x1*d1+(q2)
↦    [7]:
↦       x3
↦ [34]:
↦    [1]:
↦ 1/(q2)
↦    [2]:
↦       x2
↦    [3]:
↦       d2
↦    [4]:
↦       x3
↦    [5]:
↦       x3
↦    [6]:
↦       x3
↦    [7]:
↦       x1*x2*d1*d2-x1*d1+(q2)
↦ [35]:
↦    [1]:
↦ 1/(q2)
↦    [2]:
↦       x2
↦    [3]:
↦       x3
↦    [4]:
↦       d2
↦    [5]:
↦       x1*x2*d1*d2-x1*d1+(q2)
↦    [6]:
↦       x3
↦    [7]:
↦       x3
↦ [36]:
↦    [1]:
↦ 1/(q2)
↦    [2]:
↦       x2
↦    [3]:
↦       x3
↦    [4]:
↦       d2
↦    [5]:
```

```
↦         x3
↦      [6]:
↦         x1*x2*d1*d2-x1*d1+(q2)
↦      [7]:
↦         x3
↦   [37]:
↦      [1]:
↦   1/(q2)
↦      [2]:
↦         x2
↦      [3]:
↦         x3
↦      [4]:
↦         d2
↦      [5]:
↦         x3
↦      [6]:
↦         x3
↦      [7]:
↦         x1*x2*d1*d2-x1*d1+(q2)
↦   [38]:
↦      [1]:
↦   1/(q2)
↦      [2]:
↦         x2
↦      [3]:
↦         x3
↦      [4]:
↦         x3
↦      [5]:
↦         d2
↦      [6]:
↦         x1*x2*d1*d2-x1*d1+(q2)
↦      [7]:
↦         x3
↦   [39]:
↦      [1]:
↦   1/(q2)
↦      [2]:
↦         x2
↦      [3]:
↦         x3
↦      [4]:
↦         x3
↦      [5]:
↦         d2
↦      [6]:
↦         x3
↦      [7]:
↦         x1*x2*d1*d2-x1*d1+(q2)
↦   [40]:
↦      [1]:
↦   1/(q2)
```

```
↦     [2]:
↦        x2
↦     [3]:
↦        x3
↦     [4]:
↦        x3
↦     [5]:
↦        x3
↦     [6]:
↦        d2
↦     [7]:
↦        x1*x2*d1*d2-x1*d1+(q2)
↦  [41]:
↦     [1]:
↦  1/(q2)
↦     [2]:
↦        x3
↦     [3]:
↦        x1*x2*d1*d2-x1*d1+(q2)
↦     [4]:
↦        x2
↦     [5]:
↦        d2
↦     [6]:
↦        x3
↦     [7]:
↦        x3
↦  [42]:
↦     [1]:
↦  1/(q2)
↦     [2]:
↦        x3
↦     [3]:
↦        x1*x2*d1*d2-x1*d1+(q2)
↦     [4]:
↦        x2
↦     [5]:
↦        x3
↦     [6]:
↦        d2
↦     [7]:
↦        x3
↦  [43]:
↦     [1]:
↦  1/(q2)
↦     [2]:
↦        x3
↦     [3]:
↦        x1*x2*d1*d2-x1*d1+(q2)
↦     [4]:
↦        x2
↦     [5]:
↦        x3
```

```
↦    [6]:
↦       x3
↦    [7]:
↦       d2
↦ [44]:
↦    [1]:
↦ 1/(q2)
↦    [2]:
↦       x3
↦    [3]:
↦       x1*x2*d1*d2-x1*d1+(q2)
↦    [4]:
↦       x3
↦    [5]:
↦       x2
↦    [6]:
↦       d2
↦    [7]:
↦       x3
↦ [45]:
↦    [1]:
↦ 1/(q2)
↦    [2]:
↦       x3
↦    [3]:
↦       x1*x2*d1*d2-x1*d1+(q2)
↦    [4]:
↦       x3
↦    [5]:
↦       x2
↦    [6]:
↦       x3
↦    [7]:
↦       d2
↦ [46]:
↦    [1]:
↦ 1/(q2)
↦    [2]:
↦       x3
↦    [3]:
↦       x1*x2*d1*d2-x1*d1+(q2)
↦    [4]:
↦       x3
↦    [5]:
↦       x3
↦    [6]:
↦       x2
↦    [7]:
↦       d2
↦ [47]:
↦    [1]:
↦ 1/(q2)
↦    [2]:
```

```
↦        x3
↦     [3]:
↦        x2
↦     [4]:
↦        d2
↦     [5]:
↦        x1*x2*d1*d2-x1*d1+(q2)
↦     [6]:
↦        x3
↦     [7]:
↦        x3
↦  [48]:
↦     [1]:
↦  1/(q2)
↦     [2]:
↦        x3
↦     [3]:
↦        x2
↦     [4]:
↦        d2
↦     [5]:
↦        x3
↦     [6]:
↦        x1*x2*d1*d2-x1*d1+(q2)
↦     [7]:
↦        x3
↦  [49]:
↦     [1]:
↦  1/(q2)
↦     [2]:
↦        x3
↦     [3]:
↦        x2
↦     [4]:
↦        d2
↦     [5]:
↦        x3
↦     [6]:
↦        x3
↦     [7]:
↦        x1*x2*d1*d2-x1*d1+(q2)
↦  [50]:
↦     [1]:
↦  1/(q2)
↦     [2]:
↦        x3
↦     [3]:
↦        x2
↦     [4]:
↦        x3
↦     [5]:
↦        d2
↦     [6]:
```

```
↦       x1*x2*d1*d2-x1*d1+(q2)
↦    [7]:
↦       x3
↦ [51]:
↦    [1]:
↦ 1/(q2)
↦    [2]:
↦       x3
↦    [3]:
↦       x2
↦    [4]:
↦       x3
↦    [5]:
↦       d2
↦    [6]:
↦       x3
↦    [7]:
↦       x1*x2*d1*d2-x1*d1+(q2)
↦ [52]:
↦    [1]:
↦ 1/(q2)
↦    [2]:
↦       x3
↦    [3]:
↦       x2
↦    [4]:
↦       x3
↦    [5]:
↦       x3
↦    [6]:
↦       d2
↦    [7]:
↦       x1*x2*d1*d2-x1*d1+(q2)
↦ [53]:
↦    [1]:
↦ 1/(q2)
↦    [2]:
↦       x3
↦    [3]:
↦       x3
↦    [4]:
↦       x1*x2*d1*d2-x1*d1+(q2)
↦    [5]:
↦       x2
↦    [6]:
↦       d2
↦    [7]:
↦       x3
↦ [54]:
↦    [1]:
↦ 1/(q2)
↦    [2]:
↦       x3
```

```
↦    [3]:
↦       x3
↦    [4]:
↦       x1*x2*d1*d2-x1*d1+(q2)
↦    [5]:
↦       x2
↦    [6]:
↦       x3
↦    [7]:
↦       d2
↦ [55]:
↦    [1]:
↦ 1/(q2)
↦    [2]:
↦       x3
↦    [3]:
↦       x3
↦    [4]:
↦       x1*x2*d1*d2-x1*d1+(q2)
↦    [5]:
↦       x3
↦    [6]:
↦       x2
↦    [7]:
↦       d2
↦ [56]:
↦    [1]:
↦ 1/(q2)
↦    [2]:
↦       x3
↦    [3]:
↦       x3
↦    [4]:
↦       x2
↦    [5]:
↦       d2
↦    [6]:
↦       x1*x2*d1*d2-x1*d1+(q2)
↦    [7]:
↦       x3
↦ [57]:
↦    [1]:
↦ 1/(q2)
↦    [2]:
↦       x3
↦    [3]:
↦       x3
↦    [4]:
↦       x2
↦    [5]:
↦       d2
↦    [6]:
↦       x3
```

```
↦    [7]:
↦       x1*x2*d1*d2-x1*d1+(q2)
↦ [58]:
↦    [1]:
↦ 1/(q2)
↦    [2]:
↦       x3
↦    [3]:
↦       x3
↦    [4]:
↦       x2
↦    [5]:
↦       x3
↦    [6]:
↦       d2
↦    [7]:
↦       x1*x2*d1*d2-x1*d1+(q2)
↦ [59]:
↦    [1]:
↦ 1/(q2)
↦    [2]:
↦       x3
↦    [3]:
↦       x3
↦    [4]:
↦       x3
↦    [5]:
↦       x1*x2*d1*d2-x1*d1+(q2)
↦    [6]:
↦       x2
↦    [7]:
↦       d2
↦ [60]:
↦    [1]:
↦ 1/(q2)
↦    [2]:
↦       x3
↦    [3]:
↦       x3
↦    [4]:
↦       x3
↦    [5]:
↦       x2
↦    [6]:
↦       d2
↦    [7]:
↦       x1*x2*d1*d2-x1*d1+(q2)
```

See also: Section 7.7.14.9 [homogfacFirstQWeyl], page 542; Section 7.7.14.11 [homogfacFirstQWeyl_all], page 560.

7.7.14.11 homogfacFirstQWeyl_all

Procedure from library `ncfactor.lib` (see Section 7.7.14 [ncfactor_lib], page 534).

Usage: homogfacFirstQWeyl_all(h); h is a homogeneous polynomial in the first q-Weyl algebra with respect to the weight vector [-1,1]

Return: list

Purpose: Computes all factorizations of a homogeneous polynomial h with respect to the weight vector [-1,1] in the first q-Weyl algebra

Theory: This function is a wrapper for homogFacNthQWeyl_all. It exists to make this library downward-compatible with older versions.

Example:

```
LIB "ncfactor.lib";
ring R = (0,q),(x,d),dp;
def r = nc_algebra (q,1);
setring(r);
poly h = q^25*x^10*d^10+q^16*(q^4+q^3+q^2+q+1)^2*x^9*d^9+
q^9*(q^13+3*q^12+7*q^11+13*q^10+20*q^9+26*q^8+30*q^7+
31*q^6+26*q^5+20*q^4+13*q^3+7*q^2+3*q+1)*x^8*d^8+
q^4*(q^9+2*q^8+4*q^7+6*q^6+7*q^5+8*q^4+6*q^3+
4*q^2+2q+1)*(q^4+q^3+q^2+q+1)*(q^2+q+1)*x^7*d^7+
q*(q^2+q+1)*(q^5+2*q^4+2*q^3+3*q^2+2*q+1)*(q^4+q^3+q^2+q+1)*(q^2+1)*(q+1)*x^6*d^6+
(q^10+5*q^9+12*q^8+21*q^7+29*q^6+33*q^5+31*q^4+24*q^3+15*q^2+7*q+12)*x^5*d^5+
6*x^3*d^3+24;
homogfacFirstQWeyl_all(h);
↦ [1]:
↦    [1]:
↦ 1
↦    [2]:
↦       x5d5+6
↦    [3]:
↦       x5d5+x3d3+4
↦ [2]:
↦    [1]:
↦ 1
↦    [2]:
↦       x5d5+x3d3+4
↦    [3]:
↦       x5d5+6
```

See also: Section 7.7.14.9 [homogfacFirstQWeyl], page 542.

7.7.14.12 tst_ncfactor

Procedure from library `ncfactor.lib` (see Section 7.7.14 [ncfactor_lib], page 534).

Example:

```
LIB "ncfactor.lib";
tst_ncfactor();
↦ Testing delete_duplicates_noteval.
↦ Successful.
↦ Testing delete_duplicates_noteval_and_sort
↦ Successful.
↦ Testing combinekfinlf
↦ Successful.
↦ Testing permpp
```

```
↦ Successful.
↦ Testing binarySearch
↦ Successful.
↦ Testing getAllDivisorsFromFactList
↦ Successful.
↦ Testing triangNum
↦ Successful.
↦ Testing fromListToIntvec
↦ Successful.
↦ Testing fromIntvecToList
↦ Successful.
↦ Testing produceHomogListForProduct
↦ Successful.
↦ Testing possibleHomogPartsInBetween
↦ Successful.
↦ Testing nextSmallerEntry
↦ Successful.
↦ Testing possibleHomogPartsInBetweenNonRecursive
↦ Successful.
↦ Testing increment_intvec
↦ Successful.
↦ Testing Min
↦ Successful.
↦ Testing isListEqual
↦ Successful.
↦ Testing ncfactor_isWeyl
↦ Successful.
↦ Testing ncfactor_isQWeyl
↦ Successful.
↦ Testing checkIfProperNthWeyl
↦ Successful.
↦ Testing checkIfProperNthQWeyl
↦ Successful.
↦ Testing isInCommutativeSubRing
↦ Successful.
↦ Testing factor_commutative
↦ Successful.
↦ Testing factorizeInt
↦ Successful.
↦ Testing getAllCoeffTuplesComb
↦ Successful.
↦ Testing normalizeFactors
↦ Successful.
↦ Testing divides
↦ Successful.
↦ Testing multiplyFactIntOutput
↦ Successful.
↦ Testing testNCfac
↦ Successful.
↦ Testing monsSmallerThan
↦ Successful.
↦ Testing getMaxDegreeVec
↦ Successful.
```

```
↦ Testing isFactorizationSmaller
↦ Successful.
↦ Testing sortedInsert
↦ Successful.
↦ Testing isFactorizationEqual
↦ Successful.
↦ Testing sortFactorizations
↦ Successful.
↦ Testing homogfacNthWeyl
↦ Successful.
↦ Testing homogfacNthWeyl_all
↦ Successful.
↦ Testing facWeyl
↦ Successful.
↦ Testing checkForHomogInhomogInterchangabilityNthWeyl
↦ Successful.
↦ Testing sfacwa2NthWeyl
↦ Successful.
↦ Testing sfacwa2NthWeyl
↦ Successful.
↦ Testing determineRestOfHomogPartsNthWeyl
↦ Successful.
↦ Testing gammaForThetaNthWeyl
↦ Successful.
↦ Testing extractHomogeneousDivisorsNthWeyl
↦ Successful.
↦ Testing extractHomogeneousDivisorsLeftNthWeyl
↦ Successful.
↦ Testing extractHomogeneousDivisorsRightNthWeyl
↦ Successful.
↦ Testing computeCombinationsMinMaxHomogNthWeyl
↦ Successful.
↦ Testing getAllCombOfHomogFact
↦ Successful.
↦ Testing getPossibilitiesForRightSides
↦ Successful.
↦ Testing homogDistributionNthWeyl
↦ Successful.
↦ Testing extractLeadingTermOfNthWeylPoly
↦ Successful.
↦ Testing degreeOfNthWeylPoly
↦ Successful.
↦ Testing degreeOfNthWeylPolyInverted
↦ Successful.
↦ Testing homogwithorderNthWeyl
↦ Successful.
↦ Testing isInWeylSubAlg
↦ Successful.
↦ Testing facSubWeyl
↦ Successful.
↦ Testing facShift
↦ Successful.
↦ Testing homogfacNthQWeyl
```

```
↦ Successful.
↦ Testing homogfacNthQWeyl_all
↦ Successful.
↦ Testing ncfactor
↦ Successful.
↦ Testing ncfactor_without_opt
↦ Successful.
↦ Testing factorize_nc_s
↦ Successful.
↦ Testing delete_duplicates_noteval_and_sort
↦ All tests ran successfully.
```

7.7.15 ncpreim_lib

Library: ncpreim.lib

Purpose: Non-commutative elimination and preimage computations

Author: Daniel Andres, daniel.andres@math.rwth-aachen.de

Support: DFG Graduiertenkolleg 1632 'Experimentelle und konstruktive Algebra'

Overview: In G-algebras, elimination of variables is more involved than in the commutative case. One, not every subset of variables generates an algebra, which is again a G-algebra. Two, even if the subset of variables in question generates an admissible subalgebra, there might be no admissible elimination ordering, i.e. an elimination ordering which also satisfies the ordering condition for G-algebras.

The difference between the procedure `eliminateNC` provided in this library and the procedure `eliminate (plural)` from the kernel is that eliminateNC will always find an admissible elimination if such one exists. Moreover, the use of `slimgb` for performing Groebner basis computations is possible.

As an application of the theory of elimination, the procedure `preimageNC` is provided, which computes the preimage of an ideal under a homomorphism f: A `->` B between G-algebras A and B. In contrast to the kernel procedure `preimage (plural)`, the assumption that A is commutative is not required.

References:

(BGL) J.L. Bueso, J. Gomez-Torrecillas, F.J. Lobillo: 'Re-filtering and exactness of the Gelfand-Kirillov dimension', Bull. Sci. math. 125, 8, 689-715, 2001.
(GML) J.I. Garcia Garcia, J. Garcia Miranda, F.J. Lobillo: 'Elimination orderings and localization in PBW algebras', Linear Algebra and its Applications 430(8-9), 2133-2148, 2009.
(Lev) V. Levandovskyy: 'Intersection of ideals with non-commutative subalgebras', ISSAC'06, 212-219, ACM, 2006.

Procedures: See also: Section D.4.5 [elim_lib], page 743; Section 7.3.21 [preimage (plural)], page 348.

7.7.15.1 eliminateNC

Procedure from library `ncpreim.lib` (see Section 7.7.15 [ncpreim_lib], page 564).

Usage: eliminateNC(I,v,eng); I ideal, v intvec, eng optional int

Return: ideal, I intersected with the subring defined by the variables not index by the entries of v

Assume: The entries of v are in the range 1..nvars(basering) and the corresponding variables generate an admissible subalgebra.

Remarks: In order to determine the required elimination ordering, a linear programming problem is solved with the simplex algorithm.
Reference: (GML)
Unlike eliminate, this procedure will always find an elimination ordering, if such exists.

Note: If eng<>0, `std` is used for Groebner basis computations, otherwise (and by default) `slimgb` is used.
If printlevel=1, progress debug messages will be printed, if printlevel>=2, all the debug messages will be printed.

Example:

```
LIB "ncpreim.lib";
// (Lev): Example 2
ring r = 0,(a,b,x,d),Dp;
matrix D[4][4];
D[1,2] = 3*a; D[1,4] = 3*x^2;
D[2,3] = -x;  D[2,4] = d;     D[3,4] = 1;
def A = nc_algebra(1,D);
setring A; A;
↦ // coefficients: QQ
↦ // number of vars : 4
↦ //        block   1 : ordering Dp
↦ //                  : names    a b x d
↦ //        block   2 : ordering C
↦ // noncommutative relations:
↦ //    ba=ab+3a
↦ //    da=ad+3x2
↦ //    xb=bx-x
↦ //    db=bd+d
↦ //    dx=xd+1
ideal I = a,x;
// Since d*a-a*d = 3*x^2, any admissible ordering has to satisfy
// x^2 < a*d, while any elimination ordering for {x,d} additionally
// has to fulfil a << x and a << d.
// Hence, the weight (0,0,1,1) is not an elimination weight for
// (x,d) and the call eliminate(I,x*d); will produce an error.
eliminateNC(I,3..4);
↦ _[1]=a
// This call uses the elimination weight (0,0,1,2), which works.
```

See also: Section 7.3.5 [eliminate (plural)], page 333.

7.7.15.2 preimageNC

Procedure from library `ncpreim.lib` (see Section 7.7.15 [ncpreim_lib], page 564).

Usage: preimageNC(A,f,J[,P,eng]); A ring, f map or ideal, J ideal, P optional string, eng optional int

Assume: f defines a map from A to the basering.

Return: nothing, instead exports an object 'preim' of type ideal to ring A, being the preimage of J under f.

Note: If P is given and not equal to the empty string, the preimage is exported to A under the name specified by P.
Otherwise (and by default), P is set to 'preim'.
If eng<>0, `std` is used for Groebner basis computations, otherwise (and by default) `slimgb` is used.
If printlevel=1, progress debug messages will be printed, if printlevel>=2, all the debug messages will be printed.

Remark: Reference: (Lev)

Example:

```
LIB "ncpreim.lib";
def A = makeUgl(3); setring A; A; // universal enveloping algebra of gl_3
↦ // coefficients: QQ
↦ // number of vars : 9
↦ //        block   1 : ordering dp
↦ //                  : names    e_1_1 e_1_2 e_1_3 e_2_1 e_2_2 e_2_3 e_3_1 \
   e_3_2 e_3_3
↦ //        block   2 : ordering C
↦ // noncommutative relations:
↦ //    e_1_2e_1_1=e_1_1*e_1_2-e_1_2
↦ //    e_1_3e_1_1=e_1_1*e_1_3-e_1_3
↦ //    e_2_1e_1_1=e_1_1*e_2_1+e_2_1
↦ //    e_3_1e_1_1=e_1_1*e_3_1+e_3_1
↦ //    e_2_1e_1_2=e_1_2*e_2_1-e_1_1+e_2_2
↦ //    e_2_2e_1_2=e_1_2*e_2_2-e_1_2
↦ //    e_2_3e_1_2=e_1_2*e_2_3-e_1_3
↦ //    e_3_1e_1_2=e_1_2*e_3_1+e_3_2
↦ //    e_2_1e_1_3=e_1_3*e_2_1+e_2_3
↦ //    e_3_1e_1_3=e_1_3*e_3_1-e_1_1+e_3_3
↦ //    e_3_2e_1_3=e_1_3*e_3_2-e_1_2
↦ //    e_3_3e_1_3=e_1_3*e_3_3-e_1_3
↦ //    e_2_2e_2_1=e_2_1*e_2_2+e_2_1
↦ //    e_3_2e_2_1=e_2_1*e_3_2+e_3_1
↦ //    e_2_3e_2_2=e_2_2*e_2_3-e_2_3
↦ //    e_3_2e_2_2=e_2_2*e_3_2+e_3_2
↦ //    e_3_1e_2_3=e_2_3*e_3_1-e_2_1
↦ //    e_3_2e_2_3=e_2_3*e_3_2-e_2_2+e_3_3
↦ //    e_3_3e_2_3=e_2_3*e_3_3-e_2_3
↦ //    e_3_3e_3_1=e_3_1*e_3_3+e_3_1
↦ //    e_3_3e_3_2=e_3_2*e_3_3+e_3_2
ring r3 = 0,(x,y,z,Dx,Dy,Dz),dp;
def B = Weyl(); setring B; B;     // third Weyl algebra
↦ // coefficients: QQ
↦ // number of vars : 6
↦ //        block   1 : ordering dp
↦ //                  : names    x y z Dx Dy Dz
↦ //        block   2 : ordering C
↦ // noncommutative relations:
↦ //    Dxx=x*Dx+1
↦ //    Dyy=y*Dy+1
↦ //    Dzz=z*Dz+1
ideal ff = x*Dx,x*Dy,x*Dz,y*Dx,y*Dy,y*Dz,z*Dx,z*Dy,z*Dz;
map f = A,ff;                     // f: A -> B, e(i,j) |-> x(i)D(j)
```

```
ideal J = 0;
preimageNC(A,f,J,"K");            // compute K := ker(f)
setring A;
K;
↦ K[1]=e_2_3*e_3_2-e_2_2*e_3_3-e_2_2
↦ K[2]=e_1_3*e_3_2-e_1_2*e_3_3-e_1_2
↦ K[3]=e_2_3*e_3_1-e_2_1*e_3_3-e_2_1
↦ K[4]=e_2_2*e_3_1-e_2_1*e_3_2
↦ K[5]=e_1_3*e_3_1-e_1_1*e_3_3-e_1_1
↦ K[6]=e_1_2*e_3_1-e_1_1*e_3_2
↦ K[7]=e_1_3*e_2_2-e_1_2*e_2_3+e_1_3
↦ K[8]=e_1_3*e_2_1-e_1_1*e_2_3
↦ K[9]=e_1_2*e_2_1-e_1_1*e_2_2-e_1_1
```

See also: Section 7.3.21 [preimage (plural)], page 348.

7.7.15.3 admissibleSub

Procedure from library `ncpreim.lib` (see Section 7.7.15 [ncpreim_lib], page 564).

Usage: admissibleSub(v); v intvec

Assume: The entries of v are in the range 1..nvars(basering).

Return: int, 1 if the variables indexed by the entries of v form an admissible subalgebra, 0 otherwise

Example:

```
LIB "ncpreim.lib";
ring r = 0,(e,f,h),dp;
matrix d[3][3];
d[1,2] = -h; d[1,3] = 2*e; d[2,3] = -2*f;
def A = nc_algebra(1,d);
setring A; A; // A is U(sl_2)
↦ // coefficients: QQ
↦ // number of vars : 3
↦ //        block   1 : ordering dp
↦ //                  : names    e f h
↦ //        block   2 : ordering C
↦ // noncommutative relations:
↦ //    fe=ef-h
↦ //    he=eh+2e
↦ //    hf=fh-2f
// the subalgebra generated by e,f is not admissible since [e,f]=h
admissibleSub(1..2);
↦ 0
// but the subalgebra generated by f,h is admissible since [f,h]=2f
admissibleSub(2..3);
↦ 1
```

7.7.15.4 isUpperTriangular

Procedure from library `ncpreim.lib` (see Section 7.7.15 [ncpreim_lib], page 564).

Usage: isUpperTriangular(M[,k]); M a matrix, k an optional int

Return: int, 1 if the given matrix is upper triangular,
0 otherwise.

Note: If k<>0 is given, it is checked whether M is strictly upper triangular.

Example:

```
LIB "ncpreim.lib";
ring r = 0,x,dp;
matrix M[2][3] =
0,1,2,
0,0,3;
isUpperTriangular(M);
↦ 1
isUpperTriangular(M,1);
↦ 1
M[2,2] = 4;
isUpperTriangular(M);
↦ 1
isUpperTriangular(M,1);
↦ 0
```

7.7.15.5 appendWeight2Ord

Procedure from library `ncpreim.lib` (see Section 7.7.15 [ncpreim_lib], page 564).

Usage: appendWeight2Ord(w); w an intvec

Return: ring, the basering equipped with the ordering (a(w),<), where < is the ordering of the basering.

Example:

```
LIB "ncpreim.lib";
ring r = 0,(a,b,x,d),Dp;
intvec w = 1,2,3,4;
def r2 = appendWeight2Ord(w); // for a commutative ring
r2;
↦ // coefficients: QQ
↦ // number of vars : 4
↦ //        block   1 : ordering a
↦ //                  : names    a b x d
↦ //                  : weights  1 2 3 4
↦ //        block   2 : ordering Dp
↦ //                  : names    a b x d
↦ //        block   3 : ordering C
matrix D[4][4];
D[1,2] = 3*a;  D[1,4] = 3*x^2;  D[2,3] = -x;
D[2,4] = d;    D[3,4] = 1;
def A = nc_algebra(1,D);
setring A; A;
↦ // coefficients: QQ
↦ // number of vars : 4
↦ //        block   1 : ordering Dp
↦ //                  : names    a b x d
↦ //        block   2 : ordering C
↦ // noncommutative relations:
```

```
↦ //    ba=ab+3a
↦ //    da=ad+3x2
↦ //    xb=bx-x
↦ //    db=bd+d
↦ //    dx=xd+1
w = 2,1,1,1;
def B = appendWeight2Ord(w);  // for a non-commutative ring
setring B; B;
↦ // coefficients: QQ
↦ // number of vars : 4
↦ //        block   1 : ordering a
↦ //                  : names    a b x d
↦ //                  : weights  2 1 1 1
↦ //        block   2 : ordering Dp
↦ //                  : names    a b x d
↦ //        block   3 : ordering C
↦ // noncommutative relations:
↦ //    ba=ab+3a
↦ //    da=ad+3x2
↦ //    xb=bx-x
↦ //    db=bd+d
↦ //    dx=xd+1
```

7.7.15.6 elimWeight

Procedure from library `ncpreim.lib` (see Section 7.7.15 [ncpreim_lib], page 564).

Usage: elimWeight(v); v an intvec

Assume: The basering is a G-algebra.
The entries of v are in the range 1..nvars(basering) and the corresponding variables generate an admissible subalgebra.

Return: intvec, say w, such that the ordering (a(w),<), where < is any admissible global ordering, is an elimination ordering for the subalgebra generated by the variables indexed by the entries of the given intvec.

Note: If no such ordering exists, the zero intvec is returned.

Remark: Reference: (BGL), (GML)

Example:

```
LIB "ncpreim.lib";
// (Lev): Example 2
ring r = 0,(a,b,x,d),Dp;
matrix D[4][4];
D[1,2] = 3*a;  D[1,4] = 3*x^2;  D[2,3] = -x;
D[2,4] = d;    D[3,4] = 1;
def A = nc_algebra(1,D);
setring A; A;
↦ // coefficients: QQ
↦ // number of vars : 4
↦ //        block   1 : ordering Dp
↦ //                  : names    a b x d
↦ //        block   2 : ordering C
↦ // noncommutative relations:
```

```
↦ //    ba=ab+3a
↦ //    da=ad+3x2
↦ //    xb=bx-x
↦ //    db=bd+d
↦ //    dx=xd+1
// Since d*a-a*d = 3*x^2, any admissible ordering has to satisfy
// x^2 < a*d, while any elimination ordering for {x,d} additionally
// has to fulfil a << x and a << d.
// Hence neither a block ordering with weights
// (1,1,1,1) nor a weighted ordering with weight (0,0,1,1) will do.
intvec v = 3,4;
elimWeight(v);
↦ 0,0,1,2
```

7.7.15.7 extendedTensor

Procedure from library `ncpreim.lib` (see Section 7.7.15 [ncpreim_lib], page 564).

Usage: extendedTensor(A,I); A ring, I ideal

Return: ring, A+B (where B denotes the basering) extended with non- commutative relations between the vars of A and B, which arise from the homomorphism A -> B induced by I in the usual sense, i.e. if the vars of A are named x(i) and the vars of B y(j), then putting q(i)(j) = leadcoef(y(j)*I[i])/leadcoef(I[i]*y(j)) and r(i)(j) = y(j)*I[i] - q(i)(j)*I[i]*y(j) yields the relation y(j)*x(i) = q(i)(j)*x(i)*y(j)+r(i)(j).

Remark: Reference: (Lev)

Example:

```
LIB "ncpreim.lib";
def A = makeWeyl(2);
setring A; A;
↦ // coefficients: QQ
↦ // number of vars : 4
↦ //        block   1 : ordering dp
↦ //                  : names    x(1) x(2) D(1) D(2)
↦ //        block   2 : ordering C
↦ // noncommutative relations:
↦ //    D(1)x(1)=x(1)*D(1)+1
↦ //    D(2)x(2)=x(2)*D(2)+1
def B = makeUgl(2);
setring B; B;
↦ // coefficients: QQ
↦ // number of vars : 4
↦ //        block   1 : ordering dp
↦ //                  : names    e_1_1 e_1_2 e_2_1 e_2_2
↦ //        block   2 : ordering C
↦ // noncommutative relations:
↦ //    e_1_2e_1_1=e_1_1*e_1_2-e_1_2
↦ //    e_2_1e_1_1=e_1_1*e_2_1+e_2_1
↦ //    e_2_1e_1_2=e_1_2*e_2_1-e_1_1+e_2_2
↦ //    e_2_2e_1_2=e_1_2*e_2_2-e_1_2
↦ //    e_2_2e_2_1=e_2_1*e_2_2+e_2_1
ideal I = var(1)*var(3), var(1)*var(4), var(2)*var(3), var(2)*var(4);
I;
```

```
↦ I[1]=e_1_1*e_2_1
↦ I[2]=e_1_1*e_2_2
↦ I[3]=e_1_2*e_2_1
↦ I[4]=e_1_2*e_2_2
def C = extendedTensor(A,I);
setring C; C;
↦ // coefficients: QQ
↦ // number of vars : 8
↦ //        block   1 : ordering dp
↦ //                  : names    x(1) x(2) D(1) D(2)
↦ //        block   2 : ordering dp
↦ //                  : names    e_1_1 e_1_2 e_2_1 e_2_2
↦ //        block   3 : ordering C
↦ // noncommutative relations:
↦ //    D(1)x(1)=x(1)*D(1)+1
↦ //    e_1_1x(1)=x(1)*e_1_1-e_1_1*e_2_1
↦ //    e_1_2x(1)=x(1)*e_1_2+e_1_1^2-e_1_2*e_2_1-e_1_1*e_2_2
↦ //    e_2_1x(1)=x(1)*e_2_1+e_2_1^2
↦ //    e_2_2x(1)=x(1)*e_2_2+e_1_1*e_2_1
↦ //    D(2)x(2)=x(2)*D(2)+1
↦ //    e_1_2x(2)=x(2)*e_1_2+e_1_1*e_1_2-e_1_2*e_2_2
↦ //    e_2_1x(2)=x(2)*e_2_1-e_1_1*e_2_1+e_2_1*e_2_2
↦ //    e_1_2D(1)=D(1)*e_1_2+e_1_1*e_1_2-e_1_2*e_2_2-e_1_2
↦ //    e_2_1D(1)=D(1)*e_2_1-e_1_1*e_2_1+e_2_1*e_2_2+e_2_1
↦ //    e_1_1D(2)=D(2)*e_1_1+e_1_2*e_2_2
↦ //    e_1_2D(2)=D(2)*e_1_2+e_1_2^2
↦ //    e_2_1D(2)=D(2)*e_2_1-e_1_2*e_2_1-e_1_1*e_2_2+e_2_2^2
↦ //    e_2_2D(2)=D(2)*e_2_2-e_1_2*e_2_2
↦ //    e_1_2e_1_1=e_1_1*e_1_2-e_1_2
↦ //    e_2_1e_1_1=e_1_1*e_2_1+e_2_1
↦ //    e_2_1e_1_2=e_1_2*e_2_1-e_1_1+e_2_2
↦ //    e_2_2e_1_2=e_1_2*e_2_2-e_1_2
↦ //    e_2_2e_2_1=e_2_1*e_2_2+e_2_1
```

7.7.16 nctools_lib

Library: nctools.lib

Purpose: General tools for noncommutative algebras

Authors: Levandovskyy V., levandov@mathematik.uni-kl.de,
Lobillo, F.J., jlobillo@ugr.es,
Rabelo, C., crabelo@ugr.es,
Motsak, O., U@D, where U={motsak}, D={mathematik.uni-kl.de}

Overview: Support: DFG (Deutsche Forschungsgesellschaft) and Metodos algebraicos y efectivos en grupos cuanticos, BFM2001-3141, MCYT, Jose Gomez-Torrecillas (Main researcher).

Procedures:

7.7.16.1 Gweights

Procedure from library `nctools.lib` (see Section 7.7.16 [nctools_lib], page 571).

Usage: Gweights(r); r a ring or a square matrix

Return: intvec

Purpose: compute an appropriate weight int vector for a G-algebra, i.e., such that \foral\;i<j\;\;lm_w(d_{ij}) <_w x_i x_j.
the polynomials d_{ij} are taken from r itself, if it is of the type ring
or defined by the given square polynomial matrix

Theory: `Gweights` returns an integer vector, whose weighting should be used to redefine the G-algebra in order to get the same non-commutative structure w.r.t. a weighted ordering.
If the input is a matrix and the output is the zero vector then there is not a G-algebra structure associated to these relations with respect to the given variables.
Another possibility is to use `weightedRing` to obtain directly a G-algebra with the new appropriate (weighted) ordering.

Example:

```
LIB "nctools.lib";
ring r = (0,q),(a,b,c,d),lp;
matrix C[4][4];
C[1,2]=q; C[1,3]=q; C[1,4]=1; C[2,3]=1; C[2,4]=q; C[3,4]=q;
matrix D[4][4];
D[1,4]=(q-1/q)*b*c;
def S = nc_algebra(C,D); setring S; S;
↦ // coefficients: QQ(q)
↦ // number of vars : 4
↦ //        block   1 : ordering lp
↦ //                  : names    a b c d
↦ //        block   2 : ordering C
↦ // noncommutative relations:
↦ //    ba=(q)*ab
↦ //    ca=(q)*ac
↦ //    da=ad+(q2-1)/(q)*bc
↦ //    db=(q)*bd
↦ //    dc=(q)*cd
Gweights(S);
↦ 2,1,1,1
def D=fetch(r,D);
Gweights(D);
↦ 2,1,1,1
```

See also: Section 7.7.16.2 [weightedRing], page 572.

7.7.16.2 weightedRing

Procedure from library `nctools.lib` (see Section 7.7.16 [nctools_lib], page 571).

Usage: weightedRing(r); r a ring

Return: ring

Purpose: equip the variables of the given ring with weights such that the relations of new ring (with weighted variables) satisfies the ordering condition for G-algebras: e.g. \forall\;i<j\;\;lm_w(d_{ij})<_w x_i x_j.

Note: activate this ring with the "setring" command

Example:

```
LIB "nctools.lib";
ring r = (0,q),(a,b,c,d),lp;
matrix C[4][4];
C[1,2]=q; C[1,3]=q; C[1,4]=1; C[2,3]=1; C[2,4]=q; C[3,4]=q;
matrix D[4][4];
D[1,4]=(q-1/q)*b*c;
def S = nc_algebra(C,D); setring S; S;
↦ // coefficients: QQ(q)
↦ // number of vars : 4
↦ //        block   1 : ordering lp
↦ //                  : names    a b c d
↦ //        block   2 : ordering C
↦ // noncommutative relations:
↦ //    ba=(q)*ab
↦ //    ca=(q)*ac
↦ //    da=ad+(q2-1)/(q)*bc
↦ //    db=(q)*bd
↦ //    dc=(q)*cd
def t=weightedRing(S);
setring t; t;
↦ // coefficients: QQ(q)
↦ // number of vars : 4
↦ //        block   1 : ordering M
↦ //                  : names    a b c d
↦ //                  : weights  2 1 1 1
↦ //                  : weights  0 0 0 1
↦ //                  : weights  0 0 1 0
↦ //                  : weights  0 1 0 0
↦ //        block   2 : ordering C
↦ // noncommutative relations:
↦ //    ba=(q)*ab
↦ //    ca=(q)*ac
↦ //    da=ad+(q2-1)/(q)*bc
↦ //    db=(q)*bd
↦ //    dc=(q)*cd
```

See also: Section 7.7.16.1 [Gweights], page 571.

7.7.16.3 ndcond

Procedure from library `nctools.lib` (see Section 7.7.16 [nctools_lib], page 571).

Usage: ndcond();

Return: ideal

Purpose: compute the non-degeneracy conditions of the basering

Note: if `printlevel > 0`, the procedure displays intermediate information (by default, `printlevel`=0)

Example:

```
LIB "nctools.lib";
ring r = (0,q1,q2),(x,y,z),dp;
matrix C[3][3];
C[1,2]=q2; C[1,3]=q1; C[2,3]=1;
```

```
matrix D[3][3];
D[1,2]=x; D[1,3]=z;
def S = nc_algebra(C,D); setring S;
S;
↦ // coefficients: QQ(q1, q2)
↦ // number of vars : 3
↦ //        block   1 : ordering dp
↦ //                  : names    x y z
↦ //        block   2 : ordering C
↦ // noncommutative relations:
↦ //    yx=(q2)*x*y+x
↦ //    zx=(q1)*x*z+z
ideal j=ndcond(); // the silent version
j;
↦ j[1]=(-q2+1)*y*z-z
printlevel=1;
ideal i=ndcond(); // the verbose version
↦ Processing degree : 1
↦ 1 . 2 . 3 .
↦ failed: (-q2+1)*y*z-z
↦ done
i;
↦ i[1]=(-q2+1)*y*z-z
```

7.7.16.4 Weyl

Procedure from library `nctools.lib` (see Section 7.7.16 [nctools_lib], page 571).

Usage: Weyl()

Return: ring

Purpose: create a Weyl algebra structure on the basering

Note: Activate this ring using the command `setring`.
Assume the number of variables of a basering is 2k. (if the number of variables is odd, an error message will be returned)
by default, the procedure treats first k variables as coordinates x_i and the last k as differentials d_i
if a non-zero optional argument is given, the procedure treats 2k variables of a basering as k pairs (x_i,d_i), i.e. variables with odd numbers are treated as coordinates and with even numbers as differentials

Example:

```
LIB "nctools.lib";
ring A1=0,(x(1..2),d(1..2)),dp;
def S=Weyl();
setring S;  S;
↦ // coefficients: QQ
↦ // number of vars : 4
↦ //        block   1 : ordering dp
↦ //                  : names    x(1) x(2) d(1) d(2)
↦ //        block   2 : ordering C
↦ // noncommutative relations:
↦ //    d(1)x(1)=x(1)*d(1)+1
```

```
↦ //    d(2)x(2)=x(2)*d(2)+1
kill A1,S;
ring B1=0,(x1,d1,x2,d2),dp;
def S=Weyl(1);
setring S;  S;
↦ // coefficients: QQ
↦ // number of vars : 4
↦ //        block   1 : ordering dp
↦ //                  : names    x1 d1 x2 d2
↦ //        block   2 : ordering C
↦ // noncommutative relations:
↦ //    d1x1=x1*d1+1
↦ //    d2x2=x2*d2+1
```

See also: Section 7.7.16.5 [makeWeyl], page 575.

7.7.16.5 makeWeyl

Procedure from library `nctools.lib` (see Section 7.7.16 [nctools_lib], page 571).

Usage: makeWeyl(n,[p]); n an integer, n>0; p an optional integer (field characteristic)

Return: ring

Purpose: create the n-th Weyl algebra over the rationals Q or F_p

Note: activate this ring with the "setring" command.
The presentation of an n-th Weyl algebra is classical: D(i)x(i)=x(i)D(i)+1,
where x(i) correspond to coordinates and D(i) to partial differentiations, i=1,...,n.
If p is not prime, the next larger prime number will be used.

Example:

```
LIB "nctools.lib";
def a = makeWeyl(3);
setring a;
a;
↦ // coefficients: QQ
↦ // number of vars : 6
↦ //        block   1 : ordering dp
↦ //                  : names    x(1) x(2) x(3) D(1) D(2) D(3)
↦ //        block   2 : ordering C
↦ // noncommutative relations:
↦ //    D(1)x(1)=x(1)*D(1)+1
↦ //    D(2)x(2)=x(2)*D(2)+1
↦ //    D(3)x(3)=x(3)*D(3)+1
```

See also: Section 7.7.16.4 [Weyl], page 574.

7.7.16.6 makeHeisenberg

Procedure from library `nctools.lib` (see Section 7.7.16 [nctools_lib], page 571).

Usage: makeHeisenberg(n, [p,d]); int n (setting 2n+1 variables), optional int p (field characteristic), optional int d (power of h in the commutator)

Return: ring

Purpose: create the n-th Heisenberg algebra in the variables x(1),y(1),...,x(n),y(n),h over the rationals Q or F_p with the relations \forall\;i\in\{1,2,\ldots,n\}\;\;y(j)x(i) = x(i)y(j)+h^d.

Note: activate this ring with the `setring` command
If p is not prime, the next larger prime number will be used.

Example:

```
LIB "nctools.lib";
def a = makeHeisenberg(2);
setring a;   a;
↦ // coefficients: QQ
↦ // number of vars : 5
↦ //        block   1 : ordering lp
↦ //                  : names    x(1) x(2) y(1) y(2) h
↦ //        block   2 : ordering C
↦ // noncommutative relations:
↦ //    y(1)x(1)=x(1)*y(1)+h
↦ //    y(2)x(2)=x(2)*y(2)+h
def H3 = makeHeisenberg(3, 7, 2);
setring H3;  H3;
↦ // coefficients: ZZ/7
↦ // number of vars : 7
↦ //        block   1 : ordering lp
↦ //                  : names    x(1) x(2) x(3) y(1) y(2) y(3) h
↦ //        block   2 : ordering C
↦ // noncommutative relations:
↦ //    y(1)x(1)=x(1)*y(1)+h^2
↦ //    y(2)x(2)=x(2)*y(2)+h^2
↦ //    y(3)x(3)=x(3)*y(3)+h^2
```

See also: Section 7.7.16.5 [makeWeyl], page 575.

7.7.16.7 Exterior

Procedure from library `nctools.lib` (see Section 7.7.16 [nctools_lib], page 571).

Usage: Exterior();

Return: qring

Purpose: create the exterior algebra of a basering

Note: activate this qring with the "setring" command

Theory: given a basering, this procedure introduces the anticommutative relations x(j)x(i)=-x(i)x(j) for all j>i,
moreover, creates a factor algebra modulo the two-sided ideal, generated by x(i)^2 for all i

Example:

```
LIB "nctools.lib";
ring R = 0,(x(1..3)),dp;
def ER = Exterior();
setring ER;
ER;
↦ // coefficients: QQ
```

```
↦ // number of vars : 3
↦ //        block   1 : ordering dp
↦ //                  : names    x(1) x(2) x(3)
↦ //        block   2 : ordering C
↦ // noncommutative relations:
↦ //    x(2)x(1)=-x(1)*x(2)
↦ //    x(3)x(1)=-x(1)*x(3)
↦ //    x(3)x(2)=-x(2)*x(3)
↦ // quotient ring from ideal
↦ _[1]=x(3)^2
↦ _[2]=x(2)^2
↦ _[3]=x(1)^2
```

7.7.16.8 findimAlgebra

Procedure from library `nctools.lib` (see Section 7.7.16 [nctools_lib], page 571).

Usage: findimAlgebra(M,[r]); M a matrix, r an optional ring

Return: ring

Purpose: define a finite dimensional algebra structure on a ring

Note: the matrix M is used to define the relations x(i)*x(j) = M[i,j] in the basering (by default) or in the optional ring r.
The procedure equips the ring with the noncommutative structure.
The procedure exports the ideal (not a two-sided Groebner basis!), called `fdQuot`, for further qring definition.

Theory: finite dimensional algebra can be represented as a factor algebra of a G-algebra modulo certain two-sided ideal. The relations of a f.d. algebra are thus naturally divided into two groups: firstly, the relations on the variables of the ring, making it into G-algebra and the rest of them, which constitute the ideal which will be factored out.

Example:

```
LIB "nctools.lib";
ring r=(0,a,b),(x(1..3)),dp;
matrix S[3][3];
S[2,3]=a*x(1); S[3,2]=-b*x(1);
def A=findimAlgebra(S); setring A;
fdQuot = twostd(fdQuot);
qring Qr = fdQuot;
Qr;
↦ // coefficients: QQ(a, b)
↦ // number of vars : 3
↦ //        block   1 : ordering dp
↦ //                  : names    x(1) x(2) x(3)
↦ //        block   2 : ordering C
↦ // noncommutative relations:
↦ //    x(3)x(2)=(-b)/(a)*x(2)*x(3)
↦ // quotient ring from ideal
↦ _[1]=x(3)^2
↦ _[2]=x(2)*x(3)+(-a)*x(1)
↦ _[3]=x(1)*x(3)
↦ _[4]=x(2)^2
```

```
↦ _[5]=x(1)*x(2)
↦ _[6]=x(1)^2
```

7.7.16.9 superCommutative

Procedure from library `nctools.lib` (see Section 7.7.16 [nctools_lib], page 571).

Usage: superCommutative([b,[e, [Q]]]);

Return: qring

Purpose: create a super-commutative algebra (as a GR-algebra) over a basering,

Note: activate this qring with the "setring" command.

Note: if b==e then the resulting ring is commutative.
By default, `b=1, e=nvars(basering), Q=0`.

Theory: given a basering, this procedure introduces the anti-commutative relations
var(j)var(i)=-var(i)var(j) for all e>=j>i>=b and creates the quotient
of the anti-commutative algebra modulo the two-sided ideal, generated by
x(b)^2, ..., x(e)^2[+ Q]

Display: If `printlevel` > 1, warning debug messages will be printed

Example:

```
LIB "nctools.lib";
ring R = 0,(x(1..4)),dp; // global!
def ER = superCommutative(); // the same as Exterior (b = 1, e = N)
setring ER; ER;
↦ // coefficients: QQ
↦ // number of vars : 4
↦ //        block   1 : ordering dp
↦ //                  : names    x(1) x(2) x(3) x(4)
↦ //        block   2 : ordering C
↦ // noncommutative relations:
↦ //    x(2)x(1)=-x(1)*x(2)
↦ //    x(3)x(1)=-x(1)*x(3)
↦ //    x(4)x(1)=-x(1)*x(4)
↦ //    x(3)x(2)=-x(2)*x(3)
↦ //    x(4)x(2)=-x(2)*x(4)
↦ //    x(4)x(3)=-x(3)*x(4)
↦ // quotient ring from ideal
↦ _[1]=x(4)^2
↦ _[2]=x(3)^2
↦ _[3]=x(2)^2
↦ _[4]=x(1)^2
"Alternating variables: [", AltVarStart(), ",", AltVarEnd(), "].";
↦ Alternating variables: [ 1 , 4 ].
kill R; kill ER;
ring R = 0,(x(1..4)),(lp(1), dp(3)); // global!
def ER = superCommutative(2); // b = 2, e = N
setring ER; ER;
↦ // coefficients: QQ
↦ // number of vars : 4
↦ //        block   1 : ordering lp
↦ //                  : names    x(1)
```

```
↦ //        block   2 : ordering dp
↦ //                  : names    x(2) x(3) x(4)
↦ //        block   3 : ordering C
↦ // noncommutative relations:
↦ //    x(3)x(2)=-x(2)*x(3)
↦ //    x(4)x(2)=-x(2)*x(4)
↦ //    x(4)x(3)=-x(3)*x(4)
↦ // quotient ring from ideal
↦ _[1]=x(4)^2
↦ _[2]=x(3)^2
↦ _[3]=x(2)^2
"Alternating variables: [", AltVarStart(), ",", AltVarEnd(), "].";
↦ Alternating variables: [ 2 , 4 ].
kill R; kill ER;
ring R = 0,(x, y, z),(ds(1), dp(2)); // mixed!
def ER = superCommutative(2,3); // b = 2, e = 3
setring ER; ER;
↦ // coefficients: QQ
↦ // number of vars : 3
↦ //        block   1 : ordering ds
↦ //                  : names    x
↦ //        block   2 : ordering dp
↦ //                  : names    y z
↦ //        block   3 : ordering C
↦ // noncommutative relations:
↦ //    zy=-yz
↦ // quotient ring from ideal
↦ _[1]=y2
↦ _[2]=z2
"Alternating variables: [", AltVarStart(), ",", AltVarEnd(), "].";
↦ Alternating variables: [ 2 , 3 ].
x + 1 + z + y; // ordering on variables: y > z > 1 > x
↦ y+z+1+x
std(x - x*x*x);
↦ _[1]=x
std(ideal(x - x*x*x, x*x*z + y, z + y*x*x));
↦ _[1]=y+x2z
↦ _[2]=z+x2y
↦ _[3]=x
kill R; kill ER;
ring R = 0,(x, y, z),(ds(1), dp(2)); // mixed!
def ER = superCommutative(2, 3, ideal(x - x*x, x*x*z + y, z + y*x*x )); // b = 2, e = 3
setring ER; ER;
↦ // coefficients: QQ
↦ // number of vars : 3
↦ //        block   1 : ordering ds
↦ //                  : names    x
↦ //        block   2 : ordering dp
↦ //                  : names    y z
↦ //        block   3 : ordering C
↦ // noncommutative relations:
↦ //    zy=-yz
↦ // quotient ring from ideal
```

```
↦ _[1]=y+x2z
↦ _[2]=z+x2y
↦ _[3]=x
↦ _[4]=y2
↦ _[5]=z2
"Alternating variables: [", AltVarStart(), ",", AltVarEnd(), "].";
↦ Alternating variables: [ 2 , 3 ].
```

7.7.16.10 rightStd

Procedure from library `nctools.lib` (see Section 7.7.16 [nctools_lib], page 571).

Usage: rightStd(I); I an ideal/ module

Purpose: compute a right Groebner basis of I

Return: the same type as input

Example:

```
LIB "nctools.lib";
LIB "ncalg.lib";
def A = makeUsl(2);
setring A;
ideal I = e2,f;
option(redSB);
option(redTail);
ideal LI = std(I);
LI;
↦ LI[1]=f
↦ LI[2]=h2+h
↦ LI[3]=eh+e
↦ LI[4]=e2
ideal RI = rightStd(I);
RI;
↦ RI[1]=f
↦ RI[2]=h2-h
↦ RI[3]=eh+e
↦ RI[4]=e2
```

7.7.16.11 rightNF

Procedure from library `nctools.lib` (see Section 7.7.16 [nctools_lib], page 571).

Usage: rightNF(I); v a poly/vector, M an ideal/module

Purpose: compute a right normal form of v w.r.t. M

Return: poly/vector (as of the 1st argument)

Example:

```
LIB "nctools.lib";
LIB "ncalg.lib";
ring r = 0,(x,d),dp;
def S = nc_algebra(1,1); setring S; // Weyl algebra
ideal I = x; I = std(I);
poly  p = x*d+1;
NF(p,I); // left normal form
```

```
↦ 0
rightNF(p,I); // right normal form
↦ 1
```

7.7.16.12 rightModulo

Procedure from library `nctools.lib` (see Section 7.7.16 [nctools_lib], page 571).

Usage: rightModulo(M,N); M,N are ideals/modules

Purpose: compute a right representation of the module (M+N)/N

Return: module

Assume: M,N are presentation matrices for right modules

Example:

```
LIB "nctools.lib";
LIB "ncalg.lib";
def A = makeUsl(2);
setring A;
option(redSB);
option(redTail);
ideal I = e2,f2,h2-1;
I = twostd(I);
print(matrix(I));
↦ h2-1,fh-f,eh+e,f2,2ef-h-1,e2
ideal E  = std(e);
ideal TL = e,h-1; // the result of left modulo
TL;
↦ TL[1]=e
↦ TL[2]=h-1
ideal T = rightModulo(E,I);
T = rightStd(T+I);
T = rightStd(rightNF(T,I)); // make the output canonic
T;
↦ T[1]=h+1
↦ T[2]=e
```

7.7.16.13 moduloSlim

Procedure from library `nctools.lib` (see Section 7.7.16 [nctools_lib], page 571).

Usage: moduloSlim(A,B); A,B module/matrix/ideal

Return: module

Purpose: compute `modulo` with slimgb as engine

Example:

```
LIB "nctools.lib";
LIB "ncalg.lib";
ring r; // first classical example for modulo
ideal h1=x,y,z;    ideal h2=x;
module m=moduloSlim(h1,h2);
print(m);
↦ 1,0,0, 0,
```

```
↦ 0,0,z, x,
↦ 0,x,-y,0
// now, a noncommutative example
def A = makeUsl2(); setring A; // this algebra is U(sl_2)
ideal H2 = e2,f2,h2-1; H2 = twostd(H2);
print(matrix(H2)); // print H2 in a compact form
↦ h2-1,fh-f,eh+e,f2,2ef-h-1,e2
ideal H1 = std(e);
ideal T = moduloSlim(H1,H2);
T = std( NF(std(H2+T),H2) );
T;
↦ T[1]=h-1
↦ T[2]=e
// now, a matrix example:
ring r2 = 0,(x,d), (dp);
def R = nc_algebra(1,1); setring R;
matrix M[2][2] = d, 0, 0, d*(x*d);
matrix P[2][1] = (8x+7)*d+9x, (x2+1)*d + 5*x;
module X = moduloSlim(P,M);
print(X);
↦ 5x2d2-2xd3-5xd-6d2+5,xd5+5xd3+5d4+5d2
```

7.7.16.14 ncRelations

Procedure from library `nctools.lib` (see Section 7.7.16 [nctools_lib], page 571).

Usage: ncRelations(r); r a ring

Return: list L with two elements, both elements are of type matrix:
L[1] = matrix of coefficients C,
L[2] = matrix of polynomials D

Purpose: recover the noncommutative relations via matrices C and D from a noncommutative ring

Example:

```
LIB "nctools.lib";
ring r = 0,(x,y,z),dp;
matrix C[3][3]=0,1,2,0,0,-1,0,0,0;
print(C);
↦ 0,1,2,
↦ 0,0,-1,
↦ 0,0,0
matrix D[3][3]=0,1,2y,0,0,-2x+y+1;
print(D);
↦ 0,1,2y,
↦ 0,0,-2x+y+1,
↦ 0,0,0
def S=nc_algebra(C,D);setring S; S;
↦ // coefficients: QQ
↦ // number of vars : 3
↦ //        block   1 : ordering dp
↦ //                  : names    x y z
↦ //        block   2 : ordering C
↦ // noncommutative relations:
```

```
↦ //    yx=xy+1
↦ //    zx=2xz+2y
↦ //    zy=-yz-2x+y+1
def l=ncRelations(S);
print (l[1]);
↦ 0,1,2,
↦ 0,0,-1,
↦ 0,0,0
print (l[2]);
↦ 0,1,2y,
↦ 0,0,-2x+y+1,
↦ 0,0,0
```

See also: Section 7.4.1 [G-algebras], page 359; Section 5.1.133 [ringlist], page 251.

7.7.16.15 isCentral

Procedure from library `nctools.lib` (see Section 7.7.16 [nctools_lib], page 571).

Usage: isCentral(p); p poly

Return: int, 1 if p commutes with all variables and 0 otherwise

Purpose: check whether p is central in a basering (that is, commutes with every generator of the ring)

Note: if `printlevel > 0`, the procedure displays intermediate information (by default, `printlevel`=0)

Example:

```
LIB "nctools.lib";
ring r=0,(x,y,z),dp;
matrix D[3][3]=0;
D[1,2]=-z;
D[1,3]=2*x;
D[2,3]=-2*y;
def S = nc_algebra(1,D); setring S;
S; // this is U(sl_2)
↦ // coefficients: QQ
↦ // number of vars : 3
↦ //        block   1 : ordering dp
↦ //                  : names    x y z
↦ //        block   2 : ordering C
↦ // noncommutative relations:
↦ //    yx=xy-z
↦ //    zx=xz+2x
↦ //    zy=yz-2y
poly c = 4*x*y+z^2-2*z;
printlevel = 0;
isCentral(c);
↦ 1
poly h = x*c;
printlevel = 1;
isCentral(h);
↦ Non-central at: y
↦ Non-central at: z
↦ 0
```

7.7.16.16 isNC

Procedure from library `nctools.lib` (see Section 7.7.16 [nctools_lib], page 571).

Usage: isNC();

Purpose: check whether a basering is commutative or not

Return: int, 1 if basering is noncommutative and 0 otherwise

Example:

```
LIB "nctools.lib";
def a = makeWeyl(2);
setring a;
isNC();
↦ 1
kill a;
ring r = 17,(x(1..7)),dp;
isNC();
↦ 0
kill r;
```

7.7.16.17 isCommutative

Procedure from library `nctools.lib` (see Section 7.7.16 [nctools_lib], page 571).

Usage: isCommutative();

Return: int, 1 if basering is commutative, or 0 otherwise

Purpose: check whether basering is commutative

Example:

```
LIB "nctools.lib";
ring r = 0,(x,y),dp;
isCommutative();
↦ 1
def D = Weyl(); setring D;
isCommutative();
↦ 0
setring r;
def R = nc_algebra(1,0); setring R;
isCommutative();
↦ 1
```

7.7.16.18 isWeyl

Procedure from library `nctools.lib` (see Section 7.7.16 [nctools_lib], page 571).

Usage: isWeyl();

Return: int, 1 if basering is a Weyl algebra, or 0 otherwise

Purpose: check whether basering is a Weyl algebra

Example:

```
LIB "nctools.lib";
ring r = 0,(a,b,c,d),dp;
isWeyl();
↦ 0
def D = Weyl(1); setring D; //make from r a Weyl algebra
b*a;
↦ ab+1
isWeyl();
↦ 1
ring t = 0,(Dx,x,y,Dy),dp;
matrix M[4][4]; M[1,2]=-1; M[3,4]=1;
def T = nc_algebra(1,M); setring T;
isWeyl();
↦ 1
```

7.7.16.19 UpOneMatrix

Procedure from library `nctools.lib` (see Section 7.7.16 [nctools_lib], page 571).

Usage: UpOneMatrix(n); n an integer

Return: intmat

Purpose: compute an n x n matrix with 1's in the whole upper triangle

Note: helpful for setting noncommutative algebras with complicated coefficient matrices

Example:

```
LIB "nctools.lib";
ring   r = (0,q),(x,y,z),dp;
matrix C = UpOneMatrix(3);
C[1,3]   = q;
print(C);
↦ 0,1,(q),
↦ 0,0,1,
↦ 0,0,0
def S = nc_algebra(C,0); setring S;
S;
↦ // coefficients: QQ(q)
↦ // number of vars : 3
↦ //        block   1 : ordering dp
↦ //                  : names    x y z
↦ //        block   2 : ordering C
↦ // noncommutative relations:
↦ //    zx=(q)*xz
```

7.7.16.20 AltVarStart

Procedure from library `nctools.lib` (see Section 7.7.16 [nctools_lib], page 571).

Usage: AltVarStart();

Return: int

Purpose: returns the number of the first alternating variable of basering

Note: basering should be a super-commutative algebra constructed by
the procedure `superCommutative`, emits an error otherwise

Example:

```
LIB "nctools.lib";
ring R = 0,(x(1..4)),dp; // global!
def ER = superCommutative(2); // (b = 2, e = N)
setring ER; ER;
↦ // coefficients: QQ
↦ // number of vars : 4
↦ //        block   1 : ordering dp
↦ //                  : names    x(1) x(2) x(3) x(4)
↦ //        block   2 : ordering C
↦ // noncommutative relations:
↦ //    x(3)x(2)=-x(2)*x(3)
↦ //    x(4)x(2)=-x(2)*x(4)
↦ //    x(4)x(3)=-x(3)*x(4)
↦ // quotient ring from ideal
↦ _[1]=x(4)^2
↦ _[2]=x(3)^2
↦ _[3]=x(2)^2
"Alternating variables: [", AltVarStart(), ",", AltVarEnd(), "].";
↦ Alternating variables: [ 2 , 4 ].
setring R;
"Alternating variables: [", AltVarStart(), ",", AltVarEnd(), "].";
↦    ? SCA rings are factors by (at least) squares!
↦    ? leaving nctools.lib::AltVarStart
kill R, ER;
//////////////////////////////////////////////////////////////////
ring R = 2,(x(1..4)),dp; // the same in char. = 2!
def ER = superCommutative(2); // (b = 2, e = N)
setring ER; ER;
↦ // coefficients: ZZ/2
↦ // number of vars : 4
↦ //        block   1 : ordering dp
↦ //                  : names    x(1) x(2) x(3) x(4)
↦ //        block   2 : ordering C
↦ // quotient ring from ideal
↦ _[1]=x(4)^2
↦ _[2]=x(3)^2
↦ _[3]=x(2)^2
"Alternating variables: [", AltVarStart(), ",", AltVarEnd(), "].";
↦ Alternating variables: [ 4 , 4 ].
setring R;
"Alternating variables: [", AltVarStart(), ",", AltVarEnd(), "].";
↦    ? SCA rings are factors by (at least) squares!
↦    ? leaving nctools.lib::AltVarStart
```

7.7.16.21 AltVarEnd

Procedure from library `nctools.lib` (see Section 7.7.16 [nctools_lib], page 571).

Usage: AltVarStart();

Return: int

Purpose: returns the number of the last alternating variable of basering

Note: basering should be a super-commutative algebra constructed by the procedure `superCommutative`, emits an error otherwise

Example:

```
LIB "nctools.lib";
ring R = 0,(x(1..4)),dp; // global!
def ER = superCommutative(2); // (b = 2, e = N)
setring ER; ER;
↦ // coefficients: QQ
↦ // number of vars : 4
↦ //        block   1 : ordering dp
↦ //                  : names    x(1) x(2) x(3) x(4)
↦ //        block   2 : ordering C
↦ // noncommutative relations:
↦ //    x(3)x(2)=-x(2)*x(3)
↦ //    x(4)x(2)=-x(2)*x(4)
↦ //    x(4)x(3)=-x(3)*x(4)
↦ // quotient ring from ideal
↦ _[1]=x(4)^2
↦ _[2]=x(3)^2
↦ _[3]=x(2)^2
"Alternating variables: [", AltVarStart(), ",", AltVarEnd(), "].";
↦ Alternating variables: [ 2 , 4 ].
setring R;
"Alternating variables: [", AltVarStart(), ",", AltVarEnd(), "].";
↦    ? SCA rings are factors by (at least) squares!
↦    ? leaving nctools.lib::AltVarStart
kill R, ER;
//////////////////////////////////////////////////////////////////
ring R = 2,(x(1..4)),dp; // the same in char. = 2!
def ER = superCommutative(2); // (b = 2, e = N)
setring ER; ER;
↦ // coefficients: ZZ/2
↦ // number of vars : 4
↦ //        block   1 : ordering dp
↦ //                  : names    x(1) x(2) x(3) x(4)
↦ //        block   2 : ordering C
↦ // quotient ring from ideal
↦ _[1]=x(4)^2
↦ _[2]=x(3)^2
↦ _[3]=x(2)^2
"Alternating variables: [", AltVarStart(), ",", AltVarEnd(), "].";
↦ Alternating variables: [ 4 , 4 ].
setring R;
"Alternating variables: [", AltVarStart(), ",", AltVarEnd(), "].";
↦    ? SCA rings are factors by (at least) squares!
↦    ? leaving nctools.lib::AltVarStart
```

7.7.16.22 IsSCA

Procedure from library `nctools.lib` (see Section 7.7.16 [nctools_lib], page 571).

Usage: IsSCA();

Return: int

Purpose: returns 1 if basering is a super-commutative algebra and 0 otherwise

Example:

```
LIB "nctools.lib";
////////////////////////////////////////////////////////////////////
ring R = 0,(x(1..4)),dp; // commutative
if(IsSCA())
{ "Alternating variables: [", AltVarStart(), ",", AltVarEnd(), "]."; }
else
{ "Not a super-commutative algebra!!!"; }
↦ Not a super-commutative algebra!!!
kill R;
////////////////////////////////////////////////////////////////////
ring R = 0,(x(1..4)),dp;
def S = nc_algebra(1, 0); setring S; S; // still commutative!
↦ // coefficients: QQ
↦ // number of vars : 4
↦ //        block   1 : ordering dp
↦ //                  : names    x(1) x(2) x(3) x(4)
↦ //        block   2 : ordering C
↦ // noncommutative relations:
if(IsSCA())
{ "Alternating variables: [", AltVarStart(), ",", AltVarEnd(), "]."; }
else
{ "Not a super-commutative algebra!!!"; }
↦ Not a super-commutative algebra!!!
kill R, S;
////////////////////////////////////////////////////////////////////
ring R = 0,(x(1..4)),dp;
list CurrRing = ringlist(R);
def ER = ring(CurrRing);
setring ER; // R;
matrix E = UpOneMatrix(nvars(R));
int i, j; int b = 2; int e = 3;
for ( i = b; i < e; i++ )
{
for ( j = i+1; j <= e; j++ )
{
E[i, j] = -1;
}
}
def S = nc_algebra(E,0); setring S; S;
↦ // coefficients: QQ
↦ // number of vars : 4
↦ //        block   1 : ordering dp
↦ //                  : names    x(1) x(2) x(3) x(4)
↦ //        block   2 : ordering C
↦ // noncommutative relations:
↦ //    x(3)x(2)=-x(2)*x(3)
if(IsSCA())
{ "Alternating variables: [", AltVarStart(), ",", AltVarEnd(), "]."; }
else
{ "Not a super-commutative algebra!!!"; }
```

```
↦ Not a super-commutative algebra!!!
kill R, ER, S;
/////////////////////////////////////////////////////////////////////
ring R = 0,(x(1..4)),dp;
def ER = superCommutative(2); // (b = 2, e = N)
setring ER; ER;
↦ // coefficients: QQ
↦ // number of vars : 4
↦ //        block   1 : ordering dp
↦ //                  : names    x(1) x(2) x(3) x(4)
↦ //        block   2 : ordering C
↦ // noncommutative relations:
↦ //    x(3)x(2)=-x(2)*x(3)
↦ //    x(4)x(2)=-x(2)*x(4)
↦ //    x(4)x(3)=-x(3)*x(4)
↦ // quotient ring from ideal
↦ _[1]=x(4)^2
↦ _[2]=x(3)^2
↦ _[3]=x(2)^2
if(IsSCA())
{ "This is a SCA! Alternating variables: [", AltVarStart(), ",", AltVarEnd(), "]."; }
↦ This is a SCA! Alternating variables: [ 2 , 4 ].
else
{ "Not a super-commutative algebra!!!"; }
kill R, ER;
```

7.7.16.23 makeModElimRing

Procedure from library `nctools.lib` (see Section 7.7.16 [nctools_lib], page 571).

Usage: makeModElimRing(L); L a list

Return: ring

Purpose: create a copy of a given ring equipped with the elimination ordering for module components `(c,<)`

Note: usually the list argument contains a ring to work with

Example:

```
LIB "nctools.lib";
ring r1 = 0,(x,y,z),(C,Dp);
def r2 = makeModElimRing(r1); setring r2; r2;   kill r2;
↦ // coefficients: QQ
↦ // number of vars : 3
↦ //        block   1 : ordering c
↦ //        block   2 : ordering Dp
↦ //                  : names    x y z
ring r3 = 0,(z,t),(wp(2,3),c);
def r2 = makeModElimRing(r3); setring r2; r2; kill r2;
↦ // coefficients: QQ
↦ // number of vars : 2
↦ //        block   1 : ordering c
↦ //        block   2 : ordering wp
↦ //                  : names    z t
```

```
↦ //                  : weights  2 3
ring r4 = 0,(z,t,u,w),(a(1,2),C,wp(2,3,4,5));
def r2 = makeModElimRing(r4); setring r2; r2;
↦ // coefficients: QQ
↦ // number of vars : 4
↦ //        block   1 : ordering c
↦ //        block   2 : ordering a
↦ //                  : names    z t
↦ //                  : weights  1 2
↦ //        block   3 : ordering wp
↦ //                  : names    z t u w
↦ //                  : weights  2 3 4 5
```

7.7.16.24 embedMat

Procedure from library `nctools.lib` (see Section 7.7.16 [nctools_lib], page 571).

Usage: embedMat(A,m,n); A,B matrix/module

Return: matrix

Purpose: embed A in the left upper corner of mxn matrix

Example:

```
LIB "nctools.lib";
ring r = 0,(a,b,c,d),dp;
matrix M[2][3]; M[1,1]=a; M[1,2]=b;M[2,2]=d;M[1,3]=c;
print(M);
↦ a,b,c,
↦ 0,d,0
print(embedMat(M,3,4));
↦ a,b,c,0,
↦ 0,d,0,0,
↦ 0,0,0,0
matrix N = M; N[2,2]=0;
print(embedMat(N,3,4));
↦ a,b,c,0,
↦ 0,0,0,0,
↦ 0,0,0,0
```

7.7.17 perron_lib

Library: perron.lib

Purpose: computation of algebraic dependences

Author: Oleksandr Motsak U@D, where U={motsak}, D={mathematik.uni-kl.de}

Procedures:

7.7.17.1 perron

Procedure from library `perron.lib` (see Section 7.7.17 [perron_lib], page 590).

Usage: perron(L [, D])

Return: commutative ring with ideal ‘Relations‘

Purpose: computes polynomial relations ('Relations') between pairwise commuting polynomials of L [, up to a given degree bound D]

Note: the implementation was partially inspired by the Perron's theorem.

Example:

```
LIB "perron.lib";
int p = 3;
ring AA = p,(x,y,z),dp;
matrix D[3][3]=0;
D[1,2]=-z; D[1,3]=2*x; D[2,3]=-2*y;
def A = nc_algebra(1,D); setring A; // this algebra is U(sl_2)
ideal I = x^p, y^p, z^p-z, 4*x*y+z^2-2*z; // the center
def RA = perron( I, p );
setring RA;
RA;
↦ // coefficients: ZZ/3
↦ // number of vars : 4
↦ //        block   1 : ordering dp
↦ //                  : names    F(1) F(2) F(3) F(4)
↦ //        block   2 : ordering C
Relations; // it was exported from perron to be in the returned ring.
↦ Relations[1]=F(4)^3-F(1)*F(2)-F(3)^2+F(4)^2
// perron can be also used in a commutative case, for example:
ring B = 0,(x,y,z),dp;
ideal J = xy+z2, z2+y2, x2y2-2xy3+y4;
def RB = perron(J);
setring RB;
Relations;
↦ Relations[1]=F(1)^2-2*F(1)*F(2)+F(2)^2-F(3)
// one more test:
setring A;
map T=RA,I;
T(Relations);  // should be zero
↦ _[1]=0
```

7.7.18 qmatrix_lib

Library: qmatrix.lib

Purpose: Quantum matrices, quantum minors and symmetric groups

Authors: Lobillo, F.J., jlobillo@ugr.es
Rabelo, C., crabelo@ugr.es

Support: 'Metodos algebraicos y efectivos en grupos cuanticos', BFM2001-3141, MCYT, Jose Gomez-Torrecillas (Main researcher).

Procedures:

7.7.18.1 quantMat

Procedure from library `qmatrix.lib` (see Section 7.7.18 [qmatrix_lib], page 591).

Usage: quantMat(n [, p]); n integer (n>1), p an optional integer

Return: ring (of quantum matrices). If p is specified, the quantum parameter q will be specialized at the p-th root of unity

Purpose: compute the quantum matrix ring of order n

Note: activate this ring with the "setring" command.
The usual representation of the variables in this quantum algebra is not used because double indexes are not allowed in the variables. Instead the variables are listed by reading the rows of the usual matrix representation, that is, there will be n*n variables (one for each entry an n*N generic matrix), listed row-wise

Example:

```
LIB "qmatrix.lib";
def r = quantMat(2); // generate O_q(M_2) at q generic
setring r;   r;
↦ // coefficients: QQ(q)
↦ // number of vars : 4
↦ //        block   1 : ordering Dp
↦ //                  : names    y(1) y(2) y(3) y(4)
↦ //        block   2 : ordering C
↦ // noncommutative relations:
↦ //    y(2)y(1)=1/(q)*y(1)*y(2)
↦ //    y(3)y(1)=1/(q)*y(1)*y(3)
↦ //    y(4)y(1)=y(1)*y(4)+(-q^2+1)/(q)*y(2)*y(3)
↦ //    y(4)y(2)=1/(q)*y(2)*y(4)
↦ //    y(4)y(3)=1/(q)*y(3)*y(4)
kill r;
def r = quantMat(2,5); // generate O_q(M_2) at q^5=1
setring r;   r;
↦ // coefficients: QQ[q]/(q^4+q^3+q^2+q+1)
↦ // number of vars : 4
↦ //        block   1 : ordering Dp
↦ //                  : names    y(1) y(2) y(3) y(4)
↦ //        block   2 : ordering C
↦ // noncommutative relations:
↦ //    y(2)y(1)=(-q^3-q^2-q-1)*y(1)*y(2)
↦ //    y(3)y(1)=(-q^3-q^2-q-1)*y(1)*y(3)
↦ //    y(4)y(1)=y(1)*y(4)+(-q^3-q^2-2*q-1)*y(2)*y(3)
↦ //    y(4)y(2)=(-q^3-q^2-q-1)*y(2)*y(4)
↦ //    y(4)y(3)=(-q^3-q^2-q-1)*y(3)*y(4)
```

See also: Section 7.7.18.2 [qminor], page 592.

7.7.18.2 qminor

Procedure from library `qmatrix.lib` (see Section 7.7.18 [qmatrix_lib], page 591).

Usage: qminor(I,J,n); I,J intvec, n int

Return: poly, the quantum minor of a generic n*n quantum matrix

Assume: I is the ordered list of the rows to consider in the minor,
J is the ordered list of the columns to consider in the minor,
I and J must have the same number of elements,

n is the order of the quantum matrix algebra you are working with (quantMat(n)). The base ring should be constructed using `quantMat`.

Example:

```
LIB "qmatrix.lib";
def r = quantMat(3); // let r be a quantum matrix of order 3
setring r;
intvec u = 1,2;
intvec v = 2,3;
intvec w = 1,2,3;
qminor(w,w,3);
↦ y(1)*y(5)*y(9)+(-q)*y(1)*y(6)*y(8)+(-q)*y(2)*y(4)*y(9)+(q^2)*y(2)*y(6)*y(\
   7)+(q^2)*y(3)*y(4)*y(8)+(-q^3)*y(3)*y(5)*y(7)
qminor(u,v,3);
↦ y(2)*y(6)+(-q)*y(3)*y(5)
qminor(v,u,3);
↦ y(4)*y(8)+(-q)*y(5)*y(7)
qminor(u,u,3);
↦ y(1)*y(5)+(-q)*y(2)*y(4)
```

See also: Section 7.7.18.1 [quantMat], page 591.

7.7.18.3 SymGroup

Procedure from library `qmatrix.lib` (see Section 7.7.18 [qmatrix_lib], page 591).

Usage: SymGroup(n); n an integer (positive)

Return: intmat

Purpose: represent the symmetric group S(n) via integer vectors (permutations)

Note: each row of the output integer matrix is an element of S(n)

Example:

```
LIB "qmatrix.lib";
//  "S(3)={(1,2,3),(1,3,2),(3,1,2),(2,1,3),(2,3,1),(3,2,1)}";
SymGroup(3);
↦ 1,2,3,
↦ 1,3,2,
↦ 3,1,2,
↦ 2,1,3,
↦ 2,3,1,
↦ 3,2,1
```

See also: Section 7.7.18.5 [LengthSym], page 594; Section 7.7.18.4 [LengthSymElement], page 593.

7.7.18.4 LengthSymElement

Procedure from library `qmatrix.lib` (see Section 7.7.18 [qmatrix_lib], page 591).

Usage: LengthSymElement(v); v intvec

Return: int

Purpose: determine the length of the permutation given by v in some S(n)

Assume: v represents an element of S(n); otherwise the output may have no sense

Example:

```
LIB "qmatrix.lib";
intvec v=1,3,4,2,8,9,6,5,7,10;
LengthSymElement(v);
↦ 9
```

See also: Section 7.7.18.5 [LengthSym], page 594; Section 7.7.18.3 [SymGroup], page 593.

7.7.18.5 LengthSym

Procedure from library `qmatrix.lib` (see Section 7.7.18 [qmatrix_lib], page 591).

Usage: LengthSym(M); M an intmat

Return: intvec

Purpose: determine a vector, where the i-th element is the length of the permutation of S(n) given by the i-th row of M

Assume: M represents a subset of S(n) (each row must be an element of S(n)); otherwise, the output may have no sense

Example:

```
LIB "qmatrix.lib";
def M = SymGroup(3); M;
↦ 1,2,3,
↦ 1,3,2,
↦ 3,1,2,
↦ 2,1,3,
↦ 2,3,1,
↦ 3,2,1
LengthSym(M);
↦ 0,1,2,1,2,3
```

See also: Section 7.7.18.4 [LengthSymElement], page 593; Section 7.7.18.3 [SymGroup], page 593.

7.7.19 purityfiltration_lib

Status: experimental

Library: purityfiltration.lib

Purpose: Algorithms for computing a purity filtration of a given module

Authors: Christian Schilli, christian.schilli@rwth-aachen.de
Viktor Levandovskyy, levandov@math.rwth-aachen.de

Overview: Purity is a notion with several meanings. In our context it is equidimensionality
of a module (that is all M is pure iff any nonzero submodule of N has the same dimension as N).
Notably, one should define purity with respect to a given dimension function. In the context
of this library the corresponding function is the homological grade number j_A(M) of a module M over
an K-algebra A. j_A(M) is the minimal integer k, such that Ext^k_A(M,A) != 0.

References:

[AQ] Alban Quadrat: Grade filtration of linear functional systems, INRIA Report 7769 (2010), to appear in Acta Applicanda Mathematica.

[B93] Jan-Erik Bjoerk: Analytic D-modules and applications, Kluwer Acad. Publ., 1993.
[MB10] Mohamed Barakat: Purity Filtration and the Fine Structure of Autonomy. Proc. MTNS, 2010.

Procedures:

7.7.19.1 projectiveDimension

Procedure from library `purityfiltration.lib` (see Section 7.7.19 [purityfiltration_lib], page 594).

Usage: projectiveDimension(R,i,j), R matrix representing the Modul M=coker(R)
int i, with i=0 or i=1, j a natural number

Return: list T, a projective resolution of M and its projective dimension

Purpose: if i=0 (and by default), T[1] gives a shortest left resolution of M=D^p/D^q(R^t) and T[2] the left projective dimension of M
if i=1, T[1] gives a shortest right resolution of M=D^p/RD^q and T[2] the right projective dimension of M
in both cases T[1][j] is the (j-1)-th syzygy module of M

Note: The algorithm is due to A. Quadrat, D. Robertz, Computation of bases of free modules over the Weyl algebras, J.Symb.Comp. 42, 2007.

Example:

```
LIB "purityfiltration.lib";
// commutative example
ring D = 0,(x,y,z),dp;
matrix R[6][4]=
0,-2*x,z-2*y-x,-1,
0,z-2*x,2*y-3*x,1,
z,-6*x,-2*y-5*x,-1,
0,y-x,y-x,0,
y,-x,-y-x,0,
x,-x,-2*x,0;
// compute a left resolution of M=D^4/D^6*R
list T=projectiveDimension(transpose(R),0);
// so we have the left projective dimension
T[2];
↦ 3
//we could also compute a right resolution of M=D^6/RD^4
list T1=projectiveDimension(R,1);
// and we have right projective dimension
T1[2];
↦ 1
// check, that a syzygy matrix of R has left inverse:
print(leftInverse(syz(R)));
↦ 1,0,0,0
// so lpd(M) must be 1.
// Non-commutative example
ring D1 = 0,(x1,x2,x3,d1,d2,d3),dp;
def S=Weyl();  setring S;
matrix R[3][3]=
1/2*x2*d1, x2*d2+1, x2*d3+1/2*d1,
```

```
-1/2*x2*d2-3/2,0,1/2*d2,
-d1-1/2*x2*d3,-d2,-1/2*d3;
list T=projectiveDimension(R,0);
// left projective dimension of coker(R) is
T[2];
↦ 1
list T1=projectiveDimension(R,1);
// both modules have the same projective dimension, but different resolutions, because D is non-commutative
print(T[1][1]);
↦ 1/2*x2*d1,   -1/2*x2*d2-3/2,-1/2*x2*d3-d1,
↦ x2*d2+1,     0,             -d2,
↦ x2*d3+1/2*d1,1/2*d2,        -1/2*d3
// not the same as
print(transpose(T1[1][1]));
↦ 1/2*x2*d1,   -1/2*x2*d2-3/2,-1/2*x2*d3-d1,-1/2*x2,
↦ x2*d2+1,     0,             -d2,          0,
↦ x2*d3+1/2*d1,1/2*d2,        -1/2*d3,      1/2
```

7.7.19.2 purityFiltration

Procedure from library `purityfiltration.lib` (see Section 7.7.19 [purityfiltration_lib], page 594).

Usage: purityFiltration(S), S matrix with entries of an Auslander regular ring D

Return: a list T of two lists, purity filtration of the module M=D^q/D^p(S^t)

Purpose: the first list T[1] gives a filtration {M_i} of M,
where the i-th entry of T[1] gives the representation matrix of M_(i-1).
the second list T[2] gives representations of the factor Modules,
i.e. T[2][i] gives the repr. matrix for M_(i-1)/M_i

Example:

```
LIB "purityfiltration.lib";
ring D = 0,(x1,x2,d1,d2),dp;
def S=Weyl();
setring S;
int i;
matrix R[3][3]=0,d2-d1,d2-d1,d2,-d1,-d1-d2,d1,-d1,-2*d1;
print(R);
↦ 0, -d1+d2,-d1+d2,
↦ d2,-d1,   -d1-d2,
↦ d1,-d1,   -2*d1
list T=purityFiltration(transpose(R));
// the purity filtration of coker(M)
print(T[1][1]);
↦ 0, -d1+d2,-d1+d2,
↦ d2,-d1,   -d1-d2,
↦ d1,-d1,   -2*d1
print(T[1][2]);
↦ d2,   d2,
↦ d1-d2,0,
↦ d2,   d1
print(T[1][3]);
↦ 1,0,
↦ 0,d2,
```

```
↦ 0,d1
// factor modules of the filtration
print(T[2][1]);
↦ 0, 1,1,
↦ -1,0,1
print(T[2][2]);
↦ 1,    1,
↦ d1-d2,0
print(T[2][3]);
↦ 1,0,
↦ 0,d2,
↦ 0,d1
```

7.7.19.3 purityTriang

Procedure from library `purityfiltration.lib` (see Section 7.7.19 [purityfiltration_lib], page 594).

Usage: purityTriang(S), S matrix with entries of an Auslander regular ring D

Return: a matrix T

Purpose: compute a triangular block matrix T, such that M=D^p/D^q(S^t) is isomorphic to M'=D^p'/D^q(T^t)

Example:

```
LIB "purityfiltration.lib";
ring D = 0,(x1,x2,d1,d2),dp;
def S=Weyl();
setring S;
int i;
matrix R[3][3]=0,d2-d1,d2-d1,d2,-d1,-d1-d2,d1,-d1,-2*d1;
print(R);
↦ 0, -d1+d2,-d1+d2,
↦ d2,-d1,   -d1-d2,
↦ d1,-d1,   -2*d1
matrix T=purityTriang(transpose(R));
// a triangular blockmatrix representing the module coker(R)
print(T);
↦ 0, 1,1,-1, 0,  0, 0,
↦ -1,0,1,0,  -1, 0, 0,
↦ 0, 0,0,-d1,-d2,-1,0,
↦ 0, 0,0,-1, -1, 0, -1,
↦ 0, 0,0,0,  0,  1, -d2,
↦ 0, 0,0,0,  0,  1, 0,
↦ 0, 0,0,0,  0,  0, d1
```

7.7.19.4 gradeNumber

Procedure from library `purityfiltration.lib` (see Section 7.7.19 [purityfiltration_lib], page 594).

Usage: gradeNumber(R), R matrix, representing M=D^p/D^q(R^t) over a ring D

Return: int, grade number of M

Purpose: computes the grade number of M, i.e. the first i, with ext^i(M,D) !=0
returns -1 if M=0

Example:

```
LIB "purityfiltration.lib";
// trivial example
ring D=0,(x,y,z),dp;
matrix R[2][1]=1,x;
gradeNumber(R);
↦ 0
// R has left inverse, so M=D/D^2R=0
gradeNumber(transpose(R));
↦ -1
print(ncExt_R(0,R));
↦ 0
// so, ext^0(coker(R),D) =! 0)
//
// a little bit more complex
matrix R1[3][1]=x,-y,z;
gradeNumber(transpose(R1));
↦ 3
print(ncExt_R(0,transpose(R1)));
↦ 1
print(ncExt_R(1,transpose(R1)));
↦ 1
print(ncExt_R(2,transpose(R1)));
↦ 1,0,0,
↦ 0,1,0,
↦ 0,0,1
// ext^i are zero for i=0,1,2
matrix ext3=ncExt_R(3,transpose(R1));
print(ext3);
↦ z,y,x
// not zero
is_zero(ext3);
↦ 0
```

7.7.19.5 showgrades

Procedure from library `purityfiltration.lib` (see Section 7.7.19 [purityfiltration_lib], page 594).

Usage: showgrades(T), T list, which includes representation matrices of modules

Return: list, gradenumbers of the entries in T

Purpose: computes a list L with L[i]=gradenumber(M), M=D^p/D^qT[i]

Example:

```
LIB "purityfiltration.lib";
ring D = 0,(x,y,z),dp;
matrix R[6][4]=
0,-2*x,z-2*y-x,-1,
0,z-2*x,2*y-3*x,1,
z,-6*x,-2*y-5*x,-1,
0,y-x,y-x,0,
y,-x,-y-x,0,
x,-x,-2*x,0;
list T=purityFiltration(transpose(R))[2];
```

```
showgrades(T);
↦ [1]:
↦    0
↦ [2]:
↦    -1
↦ [3]:
↦    2
↦ [4]:
↦    3
// T[i] are i-1 pure (i=1,3,4) or zero (i=2)
```

7.7.19.6 allExtOfLeft

Procedure from library `purityfiltration.lib` (see Section 7.7.19 [purityfiltration_lib], page 594).

Usage: allExtOfLeft(M),

Return: list, entries are ext-modules

Assume: M presents a left module of finite left projective dimension n

Purpose: For a left module presented by M over the basering D,
compute a list T, whose entry T[i+1] is a matrix, presenting the right module Ext^i_D(M,D) for i=0..n

Example:

```
LIB "purityfiltration.lib";
ring D = 0,(x,y,z),dp;
matrix R[6][4]=
0,-2*x,z-2*y-x,-1,
0,z-2*x,2*y-3*x,1,
z,-6*x,-2*y-5*x,-1,
0,y-x,y-x,0,
y,-x,-y-x,0,
x,-x,-2*x,0;
// coker(R) consider the left module M=D^6/D^4R
list T=allExtOfLeft(transpose(R));
print(T[1]);
↦ 0
print(T[2]);
↦ 1,0,0,
↦ 0,1,0,
↦ 0,0,1
print(T[3]);
↦ 0,0, z,4y-z,4x-z,
↦ 0,-2,1,0,   0,
↦ 1,0, 0,0,   0,
↦ 0,1, 0,0,   0
print(T[4]);
↦ z,y,x
// right modules coker(T[i].)!!
```

7.7.19.7 allExtOfRight

Procedure from library `purityfiltration.lib` (see Section 7.7.19 [purityfiltration_lib], page 594).

Usage: allExtOfRight(R), R matrix representing the right Module M=D^q/RD^p over a ring D
M module with finite right projective dimension n

Return: list, entries are ext-modules

Purpose: computes a list T, which entries are representations of the left modules ext^i(M,D)
T[i] gives the repr. matrix of ext^(i-1)(M,D), i=1,..,n+1

Example:

```
LIB "purityfiltration.lib";
ring D = 0,(x,y,z),dp;
matrix R[6][4]=
0,-2*x,z-2*y-x,-1,
0,z-2*x,2*y-3*x,1,
z,-6*x,-2*y-5*x,-1,
0,y-x,y-x,0,
y,-x,-y-x,0,
x,-x,-2*x,0;
// coker(R) considered as right module
projectiveDimension(R,1)[2];
↦ 1
list T=allExtOfRight(R);
print(T[1]);
↦ 4x,
↦ -4y,
↦ -z,
↦ z
print(T[2]);
↦ 1,0,0,   0,0,   0,
↦ 0,0,4y-z,0,4x-z,0,
↦ 0,z,0,   y,0,   x
// left modules coker(.T[i])!!
```

7.7.19.8 doubleExt

Procedure from library `purityfiltration.lib` (see Section 7.7.19 [purityfiltration_lib], page 594).

Usage: doubleExt(R,i), R matrix representing the left Module M=D^p/D^q(R^t) over a ring D
int i, less or equal the left projective dimension of M

Return: matrix P, representing the double ext module

Purpose: computes a matrix P, which represents the left module ext^i(ext^i(M,D))

Example:

```
LIB "purityfiltration.lib";
ring D = 0,(x,y,z),dp;
matrix R[7][3]=
0 ,0,1,
1 ,-4*x+z,-z,
-1,8*x-2*z,z,
1 ,0  ,0,
0 ,x-y,0,
0 ,x-y,y,
```

```
0 ,0  ,x;
// coker(R) is 2-pure, so all doubleExt are zero
print(doubleExt(transpose(R),0));
↦ 1
print(doubleExt(transpose(R),1));
↦ 1,0,0,
↦ 0,1,0,
↦ 0,0,1
print(doubleExt(transpose(R),3));
↦ 1
// except of the second
print(doubleExt(transpose(R),2));
↦ 4y-z,4x-z
```

7.7.19.9 allDoubleExt

Procedure from library `purityfiltration.lib` (see Section 7.7.19 [purityfiltration_lib], page 594).

Usage: allDoubleExt(R), R matrix representing the left Module M=D^p/D^q(R^t) over a ring D

Return: list T, double indexed, which include all double-ext modules

Purpose: computes all double ext-modules
T[i][j] gives a representation matrix of ext^(j-1)(ext(i-1)(M,D))

Example:

```
LIB "purityfiltration.lib";
ring D = 0,(x1,x2,x3,d1,d2,d3),dp;
def S=Weyl();
setring S;
matrix R[6][4]=
0,-2*d1,d3-2*d2-d1,-1,
0,d3-2*d1,2*d2-3*d1,1,
d3,-6*d1,-2*d2-5*d1,-1,
0,d2-d1,d2-d1,0,
d2,-d1,-d2-d1,0,
d1,-d1,-2*d1,0;
list T=allDoubleExt(transpose(R));
// left projective dimension of M=coker(R) is 3
// ext^i(ext^0(M,D)), i=0,1,2,3
print(T[1][1]);
↦ 0,
↦ d1,
↦ d3,
↦ -d2
print(T[1][2]);
↦ d3,d3,d2,d1
print(T[1][3]);
↦ 1
print(T[1][4]);
↦ 1
// ext^i(ext^1(M,D)), i=0,1,2,3
print(T[2][1]);
↦ 1
```

```
print(T[2][2]);
↦ 1,0,0,
↦ 0,1,0,
↦ 0,0,1
print(T[2][3]);
↦ 0,0,0,4*d2-d3,4*d1-d3,
↦ 1,0,0,0,      0,
↦ 0,1,0,0,      0,
↦ 0,0,1,0,      0
print(T[2][4]);
↦ d3,d2,d1
// ext^i(ext^2(M,D)), i=0,1,2,3  (all zero)
print(T[3][1]);
↦ 1
print(T[3][2]);
↦ 1
print(T[3][3]);
↦ 1
print(T[3][4]);
↦ 1
// ext^i(ext^3(M,D)), i=0,1,2,3  (all zero)
print(T[4][1]);
↦ 1
print(T[4][2]);
↦ 1
print(T[4][3]);
↦ 1
print(T[4][4]);
↦ 1
```

7.7.19.10 is_pure

Procedure from library `purityfiltration.lib` (see Section 7.7.19 [purityfiltration_lib], page 594).

Usage: is_pure(R), R representing the module M=D^p/D^q(R^t)

Return: int, 0 or 1

Purpose: checks pureness of M.
returns 1, if M is pure, or 0, if it's not
remark: if M is zero, is_pure returns 1

Example:

```
LIB "purityfiltration.lib";
ring D = 0,(x,y,z),dp;
matrix R[3][2]=y,-z,x,0,0,x;
list T=purityFiltration(transpose(R));
print(transpose(std(transpose(T[2][2]))));
↦ y,-z,
↦ x,0,
↦ 0,x
// so the purity filtration of coker(R) is trivial,
// i.e. coker(R) is already pure
is_pure(transpose(R));
↦ 1
```

```
// we can also have non-pure modules:
matrix R2[6][4]=
0,-2*x,z-2*y-x,-1,
0,z-2*x,2*y-3*x,1,
z,-6*x,-2*y-5*x,-1,
0,y-x,y-x,0,
y,-x,-y-x,0,
x,-x,-2*x,0;
is_pure(transpose(R2));
↦ 0
```

7.7.19.11 purelist

Procedure from library `purityfiltration.lib` (see Section 7.7.19 [purityfiltration_lib], page 594).

Usage: purelist(T), T list, in which the i-th entry R=T[i] represents M=D^p/D^q(R^t)

Return: list M, entries of M are 0 or 1

Purpose: if T[i] is pure, M[i] is 1, else M[i] is 0

Example:

```
LIB "purityfiltration.lib";
ring D = 0,(x,y,z),dp;
matrix R[6][4]=
0,-2*x,z-2*y-x,-1,
0,z-2*x,2*y-3*x,1,
z,-6*x,-2*y-5*x,-1,
0,y-x,y-x,0,
y,-x,-y-x,0,
x,-x,-2*x,0;
is_pure(transpose(R));
↦ 0
// R is not pure, so we do the purity filtration
list T=purityFiltration(transpose(R));
// all Elements of T[2] are either zero or pure
purelist(T[2]);
↦ [1]:
↦    1
↦ [2]:
↦    1
↦ [3]:
↦    1
↦ [4]:
↦    1
```

7.7.20 nchomolog_lib

Status: experimental

Library: nchomolog.lib

Purpose: Procedures for Noncommutative Homological Algebra

Authors: Viktor Levandovskyy levandov@math.rwth-aachen.de,
Christian Schilli, christian.schilli@rwth-aachen.de,
Gerhard Pfister, pfister@mathematik.uni-kl.de

Overview: In this library we present tools of homological algebra for finitely presented modules over GR-algebras.

Procedures:

7.7.20.1 ncExt_R

Procedure from library `nchomolog.lib` (see Section 7.7.20 [nchomolog_lib], page 603).

Usage: ncExt_R(i, M); i int, M module

Compute: a presentation of Ext^i(M',R); for M'=coker(M).

Return: right module Ext, a presentation of Ext^i(M',R)

Example:

```
LIB "nchomolog.lib";
ring R     = 0,(x,y),dp;
poly F    = x2-y2;
def A = annfs(F);  setring A; // A is the 2nd Weyl algebra
matrix M[1][size(LD)] = LD; // ideal
print(M);
↦ y*Dx+x*Dy,x*Dx+y*Dy+2,x^2*Dy-y^2*Dy-2*y
print(ncExt_R(1,M)); // hence the Ext^1 is zero
↦ 1,0,
↦ 0,1
module E  = ncExt_R(2,M); // define the right module E
print(E); // E is in the opposite algebra
↦ 1, -x,    -y,
↦ Dx,y*Dy+1,x*Dy
def Aop = opposite(A);  setring Aop;
module Eop = oppose(A,E);
module T1  = ncExt_R(2,Eop);
setring A;
module T1 = oppose(Aop,T1);
print(T1); // this is a left module Ext^2(Ext^2(M,A),A)
↦ y*Dx+x*Dy,x*Dx+y*Dy+2,x^2*Dy-y^2*Dy-2*y
print(M); // it is known that M holonomic implies Ext^2(Ext^2(M,A),A) iso to M
↦ y*Dx+x*Dy,x*Dx+y*Dy+2,x^2*Dy-y^2*Dy-2*y
```

7.7.20.2 ncHom

Procedure from library `nchomolog.lib` (see Section 7.7.20 [nchomolog_lib], page 603).

Usage: ncHom(M,N); M,N modules

Compute: A presentation of Hom(M',N'), M'=coker(M), N'=coker(N)

Assume: M' is a left module, N' is a centralizing bimodule

Note: ncHom(M,N) is a right module, hence a right presentation matrix is returned

Example:

```
LIB "nchomolog.lib";
ring A=0,(x,y,z),dp;
matrix M[3][3]=1,2,3,
4,5,6,
```

```
7,8,9;
matrix N[2][2]=x,y,
z,0;
module H = ncHom(M,N);
print(H);
↦ 0,0,0,0,y,x,
↦ 0,0,0,z,0,z,
↦ 1,0,0,0,0,0,
↦ 0,1,0,0,0,0,
↦ 0,0,2,1,0,0,
↦ 0,0,1,0,0,0
```

7.7.20.3 coHom

Procedure from library `nchomolog.lib` (see Section 7.7.20 [nchomolog_lib], page 603).

Usage: coHom(A,k); A matrix, k int

Purpose: compute the matrix of a homomorphism Hom(R^k,A), where R is the basering. Let A be a matrix defining a map F1–>F2 of free R-modules, then the matrix of Hom(R^k,F1)–>Hom(R^k,F2) is computed.

Note: Both A and Hom(A,R^k) are matrices for either left or right R-module homomorphisms

Example:

```
LIB "nchomolog.lib";
ring A=0,(x,y,z),dp;
matrix M[3][3]=1,2,3,
4,5,6,
7,8,9;
module cM = coHom(M,2);
print(cM);
↦ 1,0,2,0,3,0,
↦ 0,1,0,2,0,3,
↦ 4,0,5,0,6,0,
↦ 0,4,0,5,0,6,
↦ 7,0,8,0,9,0,
↦ 0,7,0,8,0,9
```

7.7.20.4 contraHom

Procedure from library `nchomolog.lib` (see Section 7.7.20 [nchomolog_lib], page 603).

Usage: contraHom(A,k); A matrix, k int

Return: matrix

Purpose: compute the matrix of a homomorphism Hom(A,R^k), where R is the basering. Let A be a matrix defining a map F1–>F2 of free R-modules, then the matrix of Hom(F2,R^k)–>Hom(F1,R^k) is computed.

Note: if A is matrix of a left (resp. right) R-module homomorphism, then Hom(A,R^k) is a right (resp. left) R-module R-module homomorphism

Example:

```
LIB "nchomolog.lib";
ring A=0,(x,y,z),dp;
matrix M[3][3]=1,2,3,
4,5,6,
7,8,9;
module cM = contraHom(M,2);
print(cM);
↦ 1,4,7,0,0,0,
↦ 2,5,8,0,0,0,
↦ 3,6,9,0,0,0,
↦ 0,0,0,1,4,7,
↦ 0,0,0,2,5,8,
↦ 0,0,0,3,6,9
```

7.7.20.5 dmodoublext

Procedure from library `nchomolog.lib` (see Section 7.7.20 [nchomolog_lib], page 603).

Usage: dmodoublext(M [,i]); M module, i optional int

Compute: a presentation of Ext^i(Ext^i(M,D),D) for basering D

Return: left module

Note: by default, i is set to the integer part of the half of number of variables of D
for holonomic modules over Weyl algebra, the double ext is known to be holonomic left module

Example:

```
LIB "nchomolog.lib";
ring R   = 0,(x,y),dp;
poly F   = x3-y2;
def A    = annfs(F);
setring A;
dmodoublext(LD);
↦ _[1]=2*x*Dx+3*y*Dy+6
↦ _[2]=3*x^2*Dy+2*y*Dx
↦ _[3]=9*x*y*Dy^2-4*y*Dx^2+15*x*Dy
↦ _[4]=27*y^2*Dy^3+8*y*Dx^3+135*y*Dy^2+105*Dy
LD;
↦ LD[1]=2*x*Dx+3*y*Dy+6
↦ LD[2]=3*x^2*Dy+2*y*Dx
↦ LD[3]=9*x*y*Dy^2-4*y*Dx^2+15*x*Dy
↦ LD[4]=27*y^2*Dy^3+8*y*Dx^3+135*y*Dy^2+105*Dy
// fancier example:
setring A;
ideal I = Dx*(x2-y3),Dy*(x2-y3);
I = groebner(I);
print(dmodoublext(I,1));
↦ y^3-x^2
print(dmodoublext(I,2));
↦ Dy,
↦ Dx
```

7.7.20.6 is_cenBimodule

Procedure from library `nchomolog.lib` (see Section 7.7.20 [nchomolog_lib], page 603).

Usage: is_cenBimodule(M); M module

Compute: 1, if a module, presented by M can be centralizing in the sense of Artin and 0 otherwise

Note: only one condition for centralizing factor module can be checked algorithmically

Example:

```
LIB "nchomolog.lib";
def A = makeUsl2(); setring A;
poly p = 4*e*f + h^2-2*h; // generator of the center
matrix M[2][2] = p, p^2-7,0,p*(p+1);
is_cenBimodule(M); // M is centralizing
↦ 1
matrix N[2][2] = p, e*f,h,p*(p+1);
is_cenBimodule(N); // N is not centralizing
↦ 0
```

7.7.20.7 is_cenSubbimodule

Procedure from library `nchomolog.lib` (see Section 7.7.20 [nchomolog_lib], page 603).

Usage: is_cenSubbimodule(M); M module

Compute: 1, if a subbimodule, generated by the columns of M is centralizing in the sense of Artin and 0 otherwise

Example:

```
LIB "nchomolog.lib";
def A = makeUsl2(); setring A;
poly p = 4*e*f + h^2-2*h; // generator of the center
matrix M[2][2] = p, p^2-7,0,p*(p+1);
is_cenSubbimodule(M); // M is centralizing subbimodule
↦ 1
matrix N[2][2] = p, e*f,h,p*(p+1);
is_cenSubbimodule(N); // N is not centralizing subbimodule
↦ 0
```

7.7.21 ratgb_lib

Status: experimental

Library: ratgb.lib

Purpose: Groebner bases in Ore localizations of noncommutative G-algebras

Author: Viktor Levandovskyy, levandov@risc.uni-linz.ac.at

Overview: Theory: Let A be an operator algebra with `R = K[x1,.,xN]` as subring. The operators are usually denoted by `d1,..,dM`.

Assume, that A is a G-algebra, then the set `S=R-0` is multiplicatively closed Ore set in A. That is, for any s in S and a in A, there exist t in S and b in A, such that `sa=bt`. In other words, one can transform any left fraction into a right fraction. The algebra `A_S` is called an Ore localization of A with respect to S.

This library provides Groebner basis procedure for A_S, performing polynomial (that is fraction-free) computations only. Note, that there is ongoing development of the subsystem called Singular:Locapal, which will provide yet another approach to Groebner bases over such Ore localizations.

Assumptions: in order to treat such localizations constructively, some care need to be taken. We will assume that the variables `x1,...,xN` from above (which will become invertible in the localization) come as the first block among the variables of the basering. Moreover, the ordering on the basering must be an antiblock ordering, that is its matrix form has the left upper `NxN` block zero. Here is a recipe to create such an ordering easily: use 'a(w)' definitions of the ordering N times with intvecs `w_i` of the following form: `w_i` has first N components zero. The rest entries need to be positive and such, that `w1,..,wN` are linearly independent (see an example below).

Guide: with this library, it is possible
- to compute a Groebner basis of an ideal or a submodule in the 'rational' Ore localization D = A_S
- to compute a dimension of associated graded submodule (called D-dimension) - to compute a vector space dimension over Quot(R) of a submodule of D-dimension 0 (so called D-finite submodule)
- to compute a basis over Quot(R) of a D-finite submodule

Procedures: See also: Section D.11.3 [jacobson_lib], page 800.

7.7.21.1 ratstd

Procedure from library `ratgb.lib` (see Section 7.7.21 [ratgb_lib], page 607).

Usage: ratstd(I, n [,eng]); I an ideal/module, n an integer, eng an optional integer

Return: ring

Purpose: compute the Groebner basis of I in the Ore localization of the basering with respect to the subalgebra, generated by first n variables

Assume: the variables of basering are organized in two blocks and - the first block of length n contains the elements with respect to which one localizes, - the basering is equipped with the elimination block ordering for the variables in the second block

Note: the output ring C is commutative. The ideal `rGBid` in C represents the rational form of the output ideal `pGBid` in the basering. - During the computation, the D-dimension of I and the corresponding dimension as K(x)-vector space of I are computed and printed out. - Setting optional integer eng to 1, slimgb is taken as Groebner engine

Display: In order to see the steps of the computation, set printlevel to >=2

Example:

```
LIB "ratgb.lib";
ring r = (0,c),(x,y,Dx,Dy),(a(0,0,1,1),a(0,0,1,0),dp);
// this ordering is an antiblock ordering, as it must be
def S = Weyl(); setring S;
// the ideal I below annihilates parametric Appel F4 function
// where we set parameters to a=-2, b=-1 and d=0
ideal I =
x*Dx*(x*Dx+c-1) - x*(x*Dx+y*Dy-2)*(x*Dx+y*Dy-1),
y*Dy*(y*Dy-1) - y*(x*Dx+y*Dy-2)*(x*Dx+y*Dy-1);
int is = 2; // hence 1st and 2nd variables, that is x and y
```

```
// will become invertible in the localization
def A = ratstd(I,2); // main call
↦ the D-dimension is 0
↦ the K-dimension is 4
pGBid; // polynomial form of the basis in the localized ring
↦ pGBid[1]=2*x*y*Dx*Dy+x*y*Dy^2+y^2*Dy^2-y*Dy^2+(-c-2)*x*Dx-2*y*Dy+2
↦ pGBid[2]=x*Dx^2-y*Dy^2+(c)*Dx
↦ pGBid[3]=2*x^2*y*Dy^3-4*x*y^2*Dy^3+2*y^3*Dy^3-4*x*y*Dy^3-4*y^2*Dy^3+2*y*D\
   y^3+(-2*c)*x^2*Dx*Dy+(2*c)*x*Dx*Dy+2*x^2*Dy^2+(c-2)*x*y*Dy^2+(-3*c)*y^2*D\
   y^2-4*x*Dy^2+(3*c-2)*y*Dy^2+2*Dy^2+(-c^2+2*c)*x*Dx+(6*c)*y*Dy+(-6*c)
setring A; // A is a commutative ring used for presentation
rGBid; // "rational" or "localized" form of the basis
↦ rGBid[1]=(2*x*y)*Dx*Dy+(x*y+y^2-y)*Dy^2+(-c*x-2*x)*Dx+(-2*y)*Dy+2
↦ rGBid[2]=(x)*Dx^2+(-y)*Dy^2+(c)*Dx
↦ rGBid[3]=(2*x^2*y-4*x*y^2-4*x*y+2*y^3-4*y^2+2*y)*Dy^3+(-2*c*x^2+2*c*x)*Dx\
   *Dy+(c*x*y-3*c*y^2+3*c*y+2*x^2-2*x*y-4*x-2*y+2)*Dy^2+(-c^2*x+2*c*x)*Dx+(6\
   *c*y)*Dy+(-6*c)
//--- Now, let us compute a K(x,y) basis explicitly
print(matrix(kbase(rGBid)));
↦ // ** rGBid is no standard basis
↦ Dy^2,Dy,Dx,1
```

7.7.22 dmodloc_lib

Status: experimental

Library: dmodloc.lib

Purpose: Localization of algebraic D-modules and applications

Author: Daniel Andres, daniel.andres@math.rwth-aachen.de

Support: DFG Graduiertenkolleg 1632 'Experimentelle und konstruktive Algebra'

Overview: Let I be a left ideal in the n-th polynomial Weyl algebra D=K[x]<d> and let f be a polynomial in K[x].

If D/I is a holonomic module over D, it is known that the localization of D/I at f is also holonomic. The procedure `Dlocalization` computes an ideal J in D such that this localization is isomorphic to D/J as D-modules.

If one regards I as an ideal in the rational Weyl algebra as above, K(x)<d>*I, and intersects with K[x]<d>, the result is called the Weyl closure of I. The procedures `WeylClosure` (if I has finite holonomic rank) and `WeylClosure1` (if I is in the first Weyl algebra) can be used for computations.

As an application of the Weyl closure, the procedure `annRatSyz` computes a holonomic part of the annihilator of a rational function by computing certain syzygies. The full annihilator can be obtained by taking the Weyl closure of the result.

If one regards the left ideal I as system of linear PDEs, one can find its polynomial solutions with `polSol` (if I is holonomic) or `polSolFiniteRank` (if I is of finite holonomic rank). Rational solutions can be obtained with `ratSol`.

The procedure `bfctBound` computes a possible multiple of the b-function for f^s*u at a generic root of f. Here, u stands for [1] in D/I.

This library also offers the procedures `holonomicRank` and `DsingularLocus` to compute the holonomic rank and the singular locus of the D-module D/I.

References:

(OT) T. Oaku, N. Takayama: 'Algorithms for D-modules', Journal of Pure and Applied Algebra, 1998.
(OTT) T. Oaku, N. Takayama, H. Tsai: 'Polynomial and rational solutions of holonomic systems', Journal of Pure and Applied Algebra, 2001.
(OTW) T. Oaku, N. Takayama, U. Walther: 'A Localization Algorithm for D-modules', Journal of Symbolic Computation, 2000.
(Tsa) H. Tsai: 'Algorithms for algebraic analysis', PhD thesis, 2000.

Procedures: See also: Section 7.7.2 [bfun_lib], page 375; Section 7.7.4 [dmod_lib], page 400; Section 7.7.5 [dmodapp_lib], page 419; Section 7.7.6 [dmodvar_lib], page 445; Section D.6.9 [gmssing_lib], page 778.

7.7.22.1 Dlocalization

Procedure from library `dmodloc.lib` (see Section 7.7.22 [dmodloc_lib], page 609).

Usage: Dlocalization(I,f[,k,e]); I ideal, f poly, k,e optional ints

Assume: The basering is the n-th Weyl algebra over a field of characteristic 0 and for all 1<=i<=n the identity
var(i+n)*var(i)=var(i)*var(i+1)+1 holds, i.e. the sequence of variables is given by x(1),...,x(n),D(1),...,D(n), where D(i) is the differential operator belonging to x(i).
Further, assume that f does not contain any D(i) and that I is holonomic on K^n\V(f).

Return: ideal or list, computes an ideal J such that D/J is isomorphic to D/I localized at f as D-modules.
If k<>0, a list consisting of J and an integer m is returned, such that f^m represents the natural map from D/I to D/J. Otherwise (and by default), only the ideal J is returned.

Remarks: It is known that a localization at f of a holonomic D-module is again a holonomic D-module.
Reference: (OTW)

Note: If e<>0, `std` is used for Groebner basis computations, otherwise (and by default) `slimgb` is used.
If printlevel=1, progress debug messages will be printed, if printlevel>=2, all the debug messages will be printed.

Example:

```
LIB "dmodloc.lib";
// (OTW), Example 8
ring r = 0,(x,y,z,Dx,Dy,Dz),dp;
def W = Weyl();
setring W;
poly f = x^3-y^2*z^2;
ideal I = f^2*Dx+3*x^2, f^2*Dy-2*y*z^2, f^2*Dz-2*y^2*z;
// I annihilates exp(1/f)
ideal J = Dlocalization(I,f);
J;
↦ J[1]=y*Dy-z*Dz
↦ J[2]=2*y*z^2*Dx+3*x^2*Dy
↦ J[3]=2*y^2*z*Dx+3*x^2*Dz
↦ J[4]=2*z^3*Dx*Dz+3*x^2*Dy^2+2*z^2*Dx
↦ J[5]=3*y^2*z^3*Dz-2*x^4*Dx-6*x^3*z*Dz+12*y^2*z^2-12*x^3-6
```

```
↦ J[6]=4*x^4*Dx^2+12*x^3*z*Dx*Dz+9*x^2*z^2*Dz^2+40*x^3*Dx+63*x^2*z*Dz+72*x^\
   2+12*Dx
↦ J[7]=3*y*z^4*Dz^2-2*x^4*Dx*Dy-6*x^3*z*Dy*Dz+21*y*z^3*Dz-12*x^3*Dy+24*y*z^\
   2-6*Dy
↦ J[8]=3*z^5*Dz^3-2*x^4*Dx*Dy^2-6*x^3*z*Dy^2*Dz+30*z^4*Dz^2-12*x^3*Dy^2+66*\
   z^3*Dz+24*z^2-6*Dy^2
Dlocalization(I,f,1); // The natural map D/I -> D/J is given by 1/f^2
↦ [1]:
↦    _[1]=y*Dy-z*Dz
↦    _[2]=2*y*z^2*Dx+3*x^2*Dy
↦    _[3]=2*y^2*z*Dx+3*x^2*Dz
↦    _[4]=2*z^3*Dx*Dz+3*x^2*Dy^2+2*z^2*Dx
↦    _[5]=3*y^2*z^3*Dz-2*x^4*Dx-6*x^3*z*Dz+12*y^2*z^2-12*x^3-6
↦    _[6]=4*x^4*Dx^2+12*x^3*z*Dx*Dz+9*x^2*z^2*Dz^2+40*x^3*Dx+63*x^2*z*Dz+72\
   *x^2+12*Dx
↦    _[7]=3*y*z^4*Dz^2-2*x^4*Dx*Dy-6*x^3*z*Dy*Dz+21*y*z^3*Dz-12*x^3*Dy+24*y\
   *z^2-6*Dy
↦    _[8]=3*z^5*Dz^3-2*x^4*Dx*Dy^2-6*x^3*z*Dy^2*Dz+30*z^4*Dz^2-12*x^3*Dy^2+\
   66*z^3*Dz+24*z^2-6*Dy^2
↦ [2]:
↦    2
```

See also: Section 7.7.5.3 [DLoc], page 422; Section 7.7.5.5 [DLoc0], page 423; Section 7.7.5.4 [SDLoc], page 422.

7.7.22.2 WeylClosure

Procedure from library `dmodloc.lib` (see Section 7.7.22 [dmodloc_lib], page 609).

Usage: WeylClosure(I); I an ideal

Assume: The basering is the n-th Weyl algebra W over a field of characteristic 0 and for all 1<=i<=n the identity
var(i+n)*var(i)=var(i)*var(i+1)+1 holds, i.e. the sequence of variables is given by x(1),...,x(n),D(1),...,D(n), where D(i) is the differential operator belonging to x(i).
Moreover, assume that the holonomic rank of W/I is finite.

Return: ideal, the Weyl closure of I

Remarks: The Weyl closure of a left ideal I in the Weyl algebra W is defined to be the intersection of I regarded as left ideal in the rational Weyl algebra K(x(1..n))<D(1..n)> with the polynomial Weyl algebra W.
Reference: (Tsa), Algorithm 2.2.4

Note: If printlevel=1, progress debug messages will be printed, if printlevel>=2, all the debug messages will be printed.

Example:

```
LIB "dmodloc.lib";
// (OTW), Example 8
ring r = 0,(x,y,z,Dx,Dy,Dz),dp;
def D3 = Weyl();
setring D3;
poly f = x^3-y^2*z^2;
ideal I = f^2*Dx + 3*x^2, f^2*Dy-2*y*z^2, f^2*Dz-2*y^2*z;
// I annihilates exp(1/f)
```

```
WeylClosure(I);
↦ _[1]=y*Dy-z*Dz
↦ _[2]=2*y*z^2*Dx+3*x^2*Dy
↦ _[3]=2*y^2*z*Dx+3*x^2*Dz
↦ _[4]=2*z^3*Dx*Dz+3*x^2*Dy^2+2*z^2*Dx
↦ _[5]=4*x^4*Dx^2+12*x^3*z*Dx*Dz+9*x^2*z^2*Dz^2+16*x^3*Dx+27*x^2*z*Dz+12*Dx
↦ _[6]=3*y*z^4*Dz^2-2*x^4*Dx*Dy-6*x^3*z*Dy*Dz+9*y*z^3*Dz-6*Dy
↦ _[7]=3*y^2*z^3*Dz-2*x^4*Dx-6*x^3*z*Dz-6
↦ _[8]=3*z^5*Dz^3-2*x^4*Dx*Dy^2-6*x^3*z*Dy^2*Dz+18*z^4*Dz^2+18*z^3*Dz-6*Dy^\
   2
```

See also: Section 7.7.22.3 [WeylClosure1], page 612.

7.7.22.3 WeylClosure1

Procedure from library `dmodloc.lib` (see Section 7.7.22 [dmodloc_lib], page 609).

Usage: WeylClosure1(L); L a poly

Assume: The basering is the first Weyl algebra D=K<x,d|dx=xd+1> over a field K of characteristic 0.

Return: ideal, the Weyl closure of the principal left ideal generated by L

Remarks: The Weyl closure of a left ideal I in the Weyl algebra D is defined to be the intersection of I regarded as left ideal in the rational Weyl algebra K(x)<d> with the polynomial Weyl algebra D.
Reference: (Tsa), Algorithm 1.2.4

Note: If printlevel=1, progress debug messages will be printed, if printlevel>=2, all the debug messages will be printed.

Example:

```
LIB "dmodloc.lib";
ring r = 0,(x,Dx),dp;
def W = Weyl();
setring W;
poly L = (x^3+2)*Dx-3*x^2;
WeylClosure1(L);
↦ _[1]=x^3*Dx-3*x^2+2*Dx
↦ _[2]=x^2*Dx^2-2*x*Dx
↦ _[3]=x^2*Dx+Dx^2-3*x
↦ _[4]=x*Dx^2-2*Dx
L = (x^4-4*x^3+3*x^2)*Dx^2+(-6*x^3+20*x^2-12*x)*Dx+(12*x^2-32*x+12);
WeylClosure1(L);
↦ _[1]=x^4*Dx^2-4*x^3*Dx^2-6*x^3*Dx+3*x^2*Dx^2+20*x^2*Dx+12*x^2-12*x*Dx-32*\
   x+12
↦ _[2]=x^2*Dx^3-21/10*x^2*Dx^2+3/10*x*Dx^3-6/5*x*Dx^2+63/5*x*Dx-3/5*Dx^2-12\
   /5*Dx-126/5
↦ _[3]=x^3*Dx^2-43/10*x^2*Dx^2+9/10*x*Dx^3-6*x^2*Dx+12/5*x*Dx^2+109/5*x*Dx-\
   9/5*Dx^2+12*x-36/5*Dx-178/5
↦ _[4]=x^3*Dx^3-48/5*x^2*Dx^2+9/5*x*Dx^3+24/5*x*Dx^2+198/5*x*Dx-18/5*Dx^2-7\
   2/5*Dx-336/5
↦ _[5]=x^3*Dx^4-4*x^2*Dx^4+2*x^2*Dx^3+3*x*Dx^4-69/10*x^2*Dx^2+67/10*x*Dx^3-\
   24/5*x*Dx^2-3*Dx^3+207/5*x*Dx-27/5*Dx^2-18/5*Dx-414/5
↦ _[6]=x^3*Dx^6+8/3*x^3*Dx^5-4*x^2*Dx^6-2/3*x^2*Dx^5+3*x*Dx^6+16*x^2*Dx^4-2\
```

```
0*x*Dx^5-92/3*x*Dx^4+12*Dx^5+126/5*x^2*Dx^2-258/5*x*Dx^3-168/5*x*Dx^2+92/\
3*Dx^3-756/5*x*Dx+356/5*Dx^2+504/5*Dx+1512/5
```

See also: Section 7.7.22.2 [WeylClosure], page 611.

7.7.22.4 holonomicRank

Procedure from library `dmodloc.lib` (see Section 7.7.22 [dmodloc_lib], page 609).

Usage: holonomicRank(I[,e]); I ideal, e optional int

Assume: The basering is the n-th Weyl algebra over a field of characteristic 0 and for all 1<=i<=n the identity
var(i+n)*var(i)=var(i)*var(i+1)+1 holds, i.e. the sequence of variables is given by x(1),...,x(n),D(1),...,D(n), where D(i) is the differential operator belonging to x(i).

Return: int, the holonomic rank of I

Remarks: The holonomic rank of I is defined to be the K(x(1..n))-dimension of the module W/WI, where W is the rational Weyl algebra K(x(1..n))<D(1..n)>.
If this dimension is infinite, -1 is returned.

Note: If e<>0, `std` is used for Groebner basis computations, otherwise (and by default) `slimgb` is used.
If printlevel=1, progress debug messages will be printed, if printlevel>=2, all the debug messages will be printed.

Example:

```
LIB "dmodloc.lib";
// (OTW), Example 8
ring r3 = 0,(x,y,z,Dx,Dy,Dz),dp;
def D3 = Weyl();
setring D3;
poly f = x^3-y^2*z^2;
ideal I = f^2*Dx+3*x^2, f^2*Dy-2*y*z^2, f^2*Dz-2*y^2*z;
// I annihilates exp(1/f)
holonomicRank(I);
↦ 1
```

7.7.22.5 DsingularLocus

Procedure from library `dmodloc.lib` (see Section 7.7.22 [dmodloc_lib], page 609).

Usage: DsingularLocus(I); I ideal

Assume: The basering is the n-th Weyl algebra over a field of characteristic 0 and for all 1<=i<=n the identity
var(i+n)*var(i)=var(i)*var(i+1)+1 holds, i.e. the sequence of variables is given by x(1),...,x(n),D(1),...,D(n), where D(i) is the differential operator belonging to x(i).

Return: ideal, describing the singular locus of the D-module D/I

Note: If printlevel>=1, progress debug messages will be printed, if printlevel>=2, all the debug messages will be printed

Example:

```
LIB "dmodloc.lib";
// (OTW), Example 8
ring @D3 = 0,(x,y,z,Dx,Dy,Dz),dp;
def D3 = Weyl();
setring D3;
poly f = x^3-y^2*z^2;
ideal I = f^2*Dx + 3*x^2, f^2*Dy-2*y*z^2, f^2*Dz-2*y^2*z;
// I annihilates exp(1/f)
DsingularLocus(I);
↦ _[1]=y^2*z^2-x^3
```

7.7.22.6 polSol

Procedure from library `dmodloc.lib` (see Section 7.7.22 [dmodloc_lib], page 609).

Usage: polSol(I[,w,m]); I ideal, w optional intvec, m optional int

Assume: The basering is the n-th Weyl algebra W over a field of characteristic 0 and for all 1<=i<=n the identity
var(i+n)*var(i)=var(i)*var(i+1)+1 holds, i.e. the sequence of variables is given by x(1),...,x(n),D(1),...,D(n), where D(i) is the differential operator belonging to x(i).
Moreover, assume that I is holonomic.

Return: ideal, a basis of the polynomial solutions to the given system of linear PDEs with polynomial coefficients, encoded via I

Remarks: If w is given, w should consist of n strictly negative entries. Otherwise and by default, w is set to -1:n.
In this case, w is used as weight vector for the computation of a b-function.
If m is given, m is assumed to be the minimal integer root of the b-function of I w.r.t. w. Note that this assumption is not checked.
Reference: (OTT), Algorithm 2.4

Note: If printlevel=1, progress debug messages will be printed, if printlevel>=2, all the debug messages will be printed.

Example:

```
LIB "dmodloc.lib";
ring r = 0,(x,y,Dx,Dy),dp;
def W = Weyl();
setring W;
poly tx,ty = x*Dx, y*Dy;
ideal I =      // Appel F1 with parameters (2,-3,-2,5)
tx*(tx+ty+4)-x*(tx+ty+2)*(tx-3),
ty*(tx+ty+4)-y*(tx+ty+2)*(ty-2),
(x-y)*Dx*Dy+2*Dx-3*Dy;
intvec w = -1,-1;
polSol(I,w);
↦ _[1]=10*x^3*y^2-30*x^3*y-45*x^2*y^2+24*x^3+144*x^2*y+72*x*y^2-126*x^2-252\
   *x*y-42*y^2+252*x+168*y-210
```

See also: Section 7.7.22.7 [polSolFiniteRank], page 614; Section 7.7.22.8 [ratSol], page 615.

7.7.22.7 polSolFiniteRank

Procedure from library `dmodloc.lib` (see Section 7.7.22 [dmodloc_lib], page 609).

Usage: polSolFiniteRank(I[,w]); I ideal, w optional intvec

Assume: The basering is the n-th Weyl algebra W over a field of characteristic 0 and for all 1<=i<=n the identity
var(i+n)*var(i)=var(i)*var(i+1)+1 holds, i.e. the sequence of variables is given by x(1),...,x(n),D(1),...,D(n), where D(i) is the differential operator belonging to x(i).
Moreover, assume that I is of finite holonomic rank.

Return: ideal, a basis of the polynomial solutions to the given system of linear PDEs with polynomial coefficients, encoded via I

Remarks: If w is given, w should consist of n strictly negative entries. Otherwise and by default, w is set to -1:n.
In this case, w is used as weight vector for the computation of a b-function.
Reference: (OTT), Algorithm 2.6

Note: If printlevel=1, progress debug messages will be printed, if printlevel>=2, all the debug messages will be printed.

Example:

```
LIB "dmodloc.lib";
ring r = 0,(x,y,Dx,Dy),dp;
def W = Weyl();
setring W;
poly tx,ty = x*Dx, y*Dy;
ideal I =      // Appel F1 with parameters (2,-3,-2,5)
tx*(tx+ty+4)-x*(tx+ty+2)*(tx-3),
ty*(tx+ty+4)-y*(tx+ty+2)*(ty-2),
(x-y)*Dx*Dy+2*Dx-3*Dy;
intvec w = -1,-1;
polSolFiniteRank(I,w);
↦ _[1]=10*x^3*y^2-30*x^3*y-45*x^2*y^2+24*x^3+144*x^2*y+72*x*y^2-126*x^2-252\
   *x*y-42*y^2+252*x+168*y-210
```

See also: Section 7.7.22.6 [polSol], page 614; Section 7.7.22.8 [ratSol], page 615.

7.7.22.8 ratSol

Procedure from library `dmodloc.lib` (see Section 7.7.22 [dmodloc_lib], page 609).

Usage: ratSol(I); I ideal

Assume: The basering is the n-th Weyl algebra W over a field of characteristic 0 and for all 1<=i<=n the identity
var(i+n)*var(i)=var(i)*var(i+1)+1 holds, i.e. the sequence of variables is given by x(1),...,x(n),D(1),...,D(n), where D(i) is the differential operator belonging to x(i).
Moreover, assume that I is holonomic.

Return: module, a basis of the rational solutions to the given system of linear PDEs with polynomial coefficients, encoded via I Note that each entry has two components, the first one standing for the enumerator, the second one for the denominator.

Remarks: Reference: (OTT), Algorithm 3.10

Note: If printlevel=1, progress debug messages will be printed, if printlevel>=2, all the debug messages will be printed.

Example:

```
LIB "dmodloc.lib";
ring r = 0,(x,y,Dx,Dy),dp;
def W = Weyl();
setring W;
poly tx,ty = x*Dx, y*Dy;
ideal I =      // Appel F1 with parameters (3,-1,1,1) is a solution
tx*(tx+ty)-x*(tx+ty+3)*(tx-1),
ty*(tx+ty)-y*(tx+ty+3)*(ty+1);
module M = ratSol(I);
// We obtain a basis of the rational solutions to I represented by a
// module / matrix with two rows.
// Each column of the matrix represents a rational function, where
// the first row correspond to the enumerator and the second row to
// the denominator.
print(M);
↦ x-y,              x,
↦ y^4-3*y^3+3*y^2-y,y
```

See also: Section 7.7.22.6 [polSol], page 614; Section 7.7.22.7 [polSolFiniteRank], page 614.

7.7.22.9 bfctBound

Procedure from library `dmodloc.lib` (see Section 7.7.22 [dmodloc_lib], page 609).

Usage: bfctBound (I,f[,primdec]); I ideal, f poly, primdec optional string

Assume: The basering is the n-th Weyl algebra W over a field of characteristic 0 and for all 1<=i<=n the identity
var(i+n)*var(i)=var(i)*var(i+1)+1 holds, i.e. the sequence of variables is given by x(1),...,x(n),D(1),...,D(n), where D(i) is the differential operator belonging to x(i).
Moreover, assume that I is holonomic.

Return: list of roots (of type ideal) and multiplicities (of type intvec) of a multiple of the b-function for f^s*u at a generic root of f. Here, u stands for [1] in D/I.

Remarks: Reference: (OTT), Algorithm 3.4

Note: This procedure requires to compute a primary decomposition in a commutative ring. The optional string primdec can be used to specify the algorithm to do so. It may either be 'GTZ' (Gianni, Trager, Zacharias) or 'SY' (Shimoyama, Yokoyama). By default, 'GTZ' is used.
If printlevel=1, progress debug messages will be printed, if printlevel>=2, all the debug messages will be printed.

Example:

```
LIB "dmodloc.lib";
ring r = 0,(x,y,Dx,Dy),dp;
def W = Weyl();
setring W;
poly tx,ty = x*Dx, y*Dy;
ideal I =      // Appel F1 with parameters (2,-3,-2,5)
tx*(tx+ty+4)-x*(tx+ty+2)*(tx-3),
ty*(tx+ty+4)-y*(tx+ty+2)*(ty-2),
(x-y)*Dx*Dy+2*Dx-3*Dy;
kill tx,ty;
poly f = x-1;
```

```
bfctBound(I,f);
↦ [1]:
↦    _[1]=-1
↦    _[2]=-7
↦ [2]:
↦    1,1
```

See also: [bernstein], page 778; Section 7.7.2.1 [bfct], page 376; Section 7.7.2.3 [bfctAnn], page 378.

7.7.22.10 annRatSyz

Procedure from library `dmodloc.lib` (see Section 7.7.22 [dmodloc_lib], page 609).

Usage: annRatSyz(f,g[,db,eng]); f, g polynomials, db,eng optional integers

Assume: The basering is commutative and over a field of characteristic 0.

Return: ring (a Weyl algebra) containing an ideal 'LD', which is (part of) the annihilator of the rational function g/f in the corresponding Weyl algebra

Remarks: This procedure uses the computation of certain syzygies. One can obtain the full annihilator by computing the Weyl closure of the ideal LD.

Note: Activate the output ring with the `setring` command. In the output ring, the ideal 'LD' (in Groebner basis) is (part of) the annihilator of g/f.
If db>0 is given, operators of order up to db are considered, otherwise, and by default, a minimal holonomic solution is computed.
If eng<>0, `std` is used for Groebner basis computations, otherwise, and by default, `slimgb` is used.
If printlevel =1, progress debug messages will be printed, if printlevel>=2, all the debug messages will be printed.

Example:

```
LIB "dmodloc.lib";
// printlevel = 3;
ring r = 0,(x,y),dp;
poly f = 2*x*y; poly g = x^2 - y^3;
def A = annRatSyz(f,g);   // compute a holonomic solution
setring A; A;
↦ // coefficients: QQ
↦ // number of vars : 4
↦ //        block   1 : ordering dp
↦ //                  : names    x y Dx Dy
↦ //        block   2 : ordering C
↦ // noncommutative relations:
↦ //    Dxx=x*Dx+1
↦ //    Dyy=y*Dy+1
LD;
↦ LD[1]=3*x*Dx+2*y*Dy+1
↦ LD[2]=y^4*Dy-x^2*y*Dy+2*y^3+x^2
setring r;
def B = annRatSyz(f,g,5); // compute a solution up to degree 5
setring B;
LD; // this is the full annihilator as we will check below
↦ LD[1]=29223*y^2*Dx^2*Dy+2286*x*Dx*Dy^2-11464*y*Dy^3+87669*y*Dx^2+10304*Dy\
   ^2
```

```
↦ LD[2]=111*y^3*Dx^2+2760*x^2*Dx^2+1470*x*y*Dx*Dy-296*y^2*Dy^2+4050*x*Dx
↦ LD[3]=3*x*Dx+2*y*Dy+1
↦ LD[4]=y^3*Dy^2-x^2*Dy^2+6*y^2*Dy+6*y
↦ LD[5]=y^4*Dy-x^2*y*Dy+2*y^3+x^2
setring r;
def C = annRat(f,g); setring C;
LD; // the full annihilator
↦ LD[1]=3*y^2*Dx^2*Dy+2*x*Dx*Dy^2+9*y*Dx^2+4*Dy^2
↦ LD[2]=3*y^3*Dx^2-10*x*y*Dx*Dy-8*y^2*Dy^2+10*x*Dx
↦ LD[3]=y^3*Dy^2-x^2*Dy^2-6*x*y*Dx+2*y^2*Dy+4*y
↦ LD[4]=3*x*Dx+2*y*Dy+1
↦ LD[5]=y^4*Dy-x^2*y*Dy+2*y^3+x^2
ideal BLD = imap(B,LD);
NF(LD,std(BLD));
↦ _[1]=0
↦ _[2]=0
↦ _[3]=0
↦ _[4]=0
↦ _[5]=0
```

See also: Section 7.7.5.1 [annPoly], page 420; Section 7.7.5.2 [annRat], page 421.

7.7.22.11 dmodGeneralAssumptionCheck

Procedure from library `dmodloc.lib` (see Section 7.7.22 [dmodloc_lib], page 609).

Usage: dmodGeneralAssumptionCheck();

Return: nothing, but checks general assumptions on the basering

Note: This procedure checks the following conditions on the basering R and prints an error message if any of them is violated:
- R is the n-th Weyl algebra over a field of characteristic 0,
- R is not a qring,
- for all 1<=i<=n the identity var(i+n)*var(i)=var(i)*var(i+1)+1 holds, i.e. the sequence of variables is given by
x(1),...,x(n),D(1),...,D(n), where D(i) is the differential operator belonging to x(i).

Example:

```
LIB "dmodloc.lib";
ring r = 0,(x,D),dp;
dmodGeneralAssumptionCheck(); // prints error message
↦    ? Basering is not a Weyl algebra
↦    ? leaving dmodloc.lib::dmodGeneralAssumptionCheck
def W = Weyl();
setring W;
dmodGeneralAssumptionCheck(); // returns nothing
```

7.7.22.12 extendWeyl

Procedure from library `dmodloc.lib` (see Section 7.7.22 [dmodloc_lib], page 609).

Usage: extendWeyl(S); S string or list of strings

Assume: The basering is the n-th Weyl algebra over a field of characteristic 0 and for all 1<=i<=n the identity var(i+n)*var(i)=var(i)*var(i+1)+1 holds, i.e. the sequence of variables is given by x(1),...,x(n),D(1),...,D(n), where D(i) is the differential operator belonging to x(i).

Return: ring, Weyl algebra extended by vars given by S

Example:

```
LIB "dmodloc.lib";
ring @D2 = 0,(x,y,Dx,Dy),dp;
def D2 = Weyl();
setring D2;
def D3 = extendWeyl("t");
setring D3; D3;
↦ // coefficients: QQ
↦ // number of vars : 6
↦ //        block   1 : ordering dp
↦ //                  : names    t x y Dt Dx Dy
↦ //        block   2 : ordering C
↦ // noncommutative relations:
↦ //    Dtt=t*Dt+1
↦ //    Dxx=x*Dx+1
↦ //    Dyy=y*Dy+1
list L = "u","v";
def D5 = extendWeyl(L);
setring D5;
D5;
↦ // coefficients: QQ
↦ // number of vars : 10
↦ //        block   1 : ordering dp
↦ //                  : names    u v t x y Du Dv Dt Dx Dy
↦ //        block   2 : ordering C
↦ // noncommutative relations:
↦ //    Duu=u*Du+1
↦ //    Dvv=v*Dv+1
↦ //    Dtt=t*Dt+1
↦ //    Dxx=x*Dx+1
↦ //    Dyy=y*Dy+1
```

7.7.22.13 polyVars

Procedure from library `dmodloc.lib` (see Section 7.7.22 [dmodloc_lib], page 609).

Usage: polyVars(f,v); f poly, v intvec

Return: int, 1 if f contains only variables indexed by v, 0 otherwise

Example:

```
LIB "dmodloc.lib";
ring r = 0,(x,y,z),dp;
poly f = y^2+zy;
intvec v = 1,2;
polyVars(f,v); // does f depend only on x,y?
↦ 0
v = 2,3;
polyVars(f,v); // does f depend only on y,z?
↦ 1
```

7.7.22.14 monomialInIdeal

Procedure from library `dmodloc.lib` (see Section 7.7.22 [dmodloc_lib], page 609).

Usage: monomialInIdeal(I); I ideal

Return: ideal consisting of all monomials appearing in generators of ideal

Examlpe: example monomialInIdeal; shows examples

Example:

```
LIB "dmodloc.lib";
ring r = 0,(x,y),dp;
ideal I = x2+5x3y7, x-x2-6xy;
monomialInIdeal(I);
↦ _[1]=x3y7
↦ _[2]=x2
↦ _[3]=xy
↦ _[4]=x
```

7.7.22.15 vars2pars

Procedure from library `dmodloc.lib` (see Section 7.7.22 [dmodloc_lib], page 609).

Usage: vars2pars(v); v intvec

Assume: The basering is commutative.

Return: ring with variables specified by v converted into parameters

Example:

```
LIB "dmodloc.lib";
ring r = 0,(x,y,z,a,b,c),dp;
intvec v = 4,5,6;
def R = vars2pars(v);
setring R;
R;
↦ // coefficients: QQ(a, b, c)
↦ // number of vars : 3
↦ //        block   1 : ordering dp
↦ //                  : names    x y z
↦ //        block   2 : ordering C
v = 1,2;
def RR = vars2pars(v);
setring RR;
RR;
↦ // coefficients: QQ(a, b, c, x, y)
↦ // number of vars : 1
↦ //        block   1 : ordering dp
↦ //                  : names    z
↦ //        block   2 : ordering C
```

7.7.22.16 minIntRoot2

Procedure from library `dmodloc.lib` (see Section 7.7.22 [dmodloc_lib], page 609).

Usage: minIntRoot2(L); L list

Assume: L is the output of bFactor.

Return: int, the minimal integer root in a list of roots

Example:

```
LIB "dmodloc.lib";
ring r = 0,x,dp;
poly f = x*(x+1)*(x-2)*(x-5/2)*(x+5/2);
list L = bFactor(f);
minIntRoot2(L);
↦ -1
```

See also: Section 7.7.5.24 [bFactor], page 440; Section 7.7.22.17 [maxIntRoot], page 621; Section 7.7.4.21 [minIntRoot], page 417.

7.7.22.17 maxIntRoot

Procedure from library `dmodloc.lib` (see Section 7.7.22 [dmodloc_lib], page 609).

Usage: maxIntRoot(L); L list

Assume: L is the output of bFactor.

Return: int, the maximal integer root in a list of roots

Example:

```
LIB "dmodloc.lib";
ring r = 0,x,dp;
poly f = x*(x+1)*(x-2)*(x-5/2)*(x+5/2);
list L = bFactor(f);
maxIntRoot(L);
↦ 2
```

See also: Section 7.7.5.24 [bFactor], page 440; Section 7.7.22.16 [minIntRoot2], page 620.

7.7.22.18 dmodAction

Procedure from library `dmodloc.lib` (see Section 7.7.22 [dmodloc_lib], page 609).

Usage: dmodAction(id,f[,v]); id ideal or poly, f poly, v optional intvec

Assume: If v is not given, the basering is the n-th Weyl algebra W over a field of characteristic 0 and for all 1<=i<=n the identity var(i+n)*var(i)=var(i)*var(i+1)+1 holds, i.e. the sequence of variables is given by x(1),...,x(n),D(1),...,D(n), where D(i) is the differential operator belonging to x(i).
Otherwise, v is assumed to specify positions of variables, which form a Weyl algebra as a subalgebra of the basering:
If size(v) equals 2*n, then bracket(var(v[i]),var(v[j])) must equal 1 if and only if j equals i+n, and 0 otherwise, for all 1<=i,j<=n.
Further, assume that f does not contain any D(i).

Return: same type as id, the result of the natural D-module action of id on f

Note: The assumptions made are not checked.

Example:

```
LIB "dmodloc.lib";
ring r = 0,(x,y,z),dp;
poly f = x^2*z - y^3;
def A = annPoly(f);
setring A;
poly f = imap(r,f);
dmodAction(LD,f);
↦ _[1]=0
↦ _[2]=0
↦ _[3]=0
↦ _[4]=0
↦ _[5]=0
↦ _[6]=0
↦ _[7]=0
↦ _[8]=0
↦ _[9]=0
↦ _[10]=0
↦ _[11]=0
↦ _[12]=0
↦ _[13]=0
poly P = y*Dy+3*z*Dz-3;
dmodAction(P,f);
↦ 0
dmodAction(P[1],f);
↦ -3*y^3
```

7.7.22.19 dmodActionRat

Procedure from library `dmodloc.lib` (see Section 7.7.22 [dmodloc_lib], page 609).

Usage: dmodActionRat(id,w); id ideal or poly, f vector

Assume: The basering is the n-th Weyl algebra W over a field of characteristic 0 and for all 1<=i<=n the identity
var(i+n)*var(i)=var(i)*var(i+1)+1 holds, i.e. the sequence of variables is given by x(1),...,x(n),D(1),...,D(n), where D(i) is the differential operator belonging to x(i).
Further, assume that w has exactly two components, second one not 0, and that w does not contain any D(i).

Return: same type as id, the result of the natural D-module action of id on the rational function w[1]/w[2]

Example:

```
LIB "dmodloc.lib";
ring r = 0,(x,y),dp;
poly f = 2*x;  poly g = y;
def A = annRat(f,g); setring A;
poly f = imap(r,f); poly g = imap(r,g);
vector v = [f,g]; // represents f/g
// x and y act by multiplication
dmodActionRat(x,v);
↦ _[1]=2*x^2*gen(1)+y*gen(2)
dmodActionRat(y,v);
↦ _[1]=2*x*gen(1)+gen(2)
// Dx and Dy act by partial derivation
```

```
dmodActionRat(Dx,v);
↦ _[1]=y*gen(2)+2*gen(1)
dmodActionRat(Dy,v);
↦ _[1]=y^2*gen(2)-2*x*gen(1)
dmodActionRat(x*Dx+y*Dy,v);
↦ _[1]=gen(2)
setring r;
f = 2*x*y; g = x^2 - y^3;
def B = annRat(f,g); setring B;
poly f = imap(r,f); poly g = imap(r,g);
vector v = [f,g];
dmodActionRat(LD,v); // hence LD is indeed the annihilator of f/g
↦ _[1]=gen(2)
↦ _[2]=gen(2)
↦ _[3]=gen(2)
↦ _[4]=gen(2)
↦ _[5]=gen(2)
```

7.7.22.20 simplifyRat

Procedure from library `dmodloc.lib` (see Section 7.7.22 [dmodloc_lib], page 609).

Usage: simplifyRat(v); v vector

Assume: Assume that v has exactly two components, second one not 0.

Return: vector, representing simplified rational function v[1]/v[2]

Note: Possibly present non-commutative relations of the basering are ignored.

Example:

```
LIB "dmodloc.lib";
ring r = 0,(x,y),dp;
vector v = [x2-1,x+1];
simplifyRat(v);
↦ x*gen(1)+gen(2)-gen(1)
simplifyRat(v) - [x-1,1];
↦ 0
```

7.7.22.21 addRat

Procedure from library `dmodloc.lib` (see Section 7.7.22 [dmodloc_lib], page 609).

Usage: addRat(v,w); v,w vectors

Assume: Assume that v,w have exactly two components, second ones not 0.

Return: vector, representing rational function (v[1]/v[2])+(w[1]/w[2])

Note: Possibly present non-commutative relations of the basering are ignored.

Example:

```
LIB "dmodloc.lib";
ring r = 0,(x,y),dp;
vector v = [x,y];
vector w = [y,x];
addRat(v,w);
```

```
↦ x2*gen(1)+xy*gen(2)+y2*gen(1)
addRat(v,w) - [x2+y2,xy];
↦ 0
```

7.7.22.22 multRat

Procedure from library `dmodloc.lib` (see Section 7.7.22 [dmodloc_lib], page 609).

Usage: multRat(v,w); v,w vectors

Assume: Assume that v,w have exactly two components, second ones not 0.

Return: vector, representing rational function (v[1]/v[2])*(w[1]/w[2])

Note: Possibly present non-commutative relations of the basering are ignored.

Example:

```
LIB "dmodloc.lib";
ring r = 0,(x,y),dp;
vector v = [x,y];
vector w = [y,x];
multRat(v,w);
↦ gen(2)+gen(1)
multRat(v,w) - [1,1];
↦ 0
```

7.7.22.23 diffRat

Procedure from library `dmodloc.lib` (see Section 7.7.22 [dmodloc_lib], page 609).

Usage: diffRat(v,j); v vector, j int

Assume: Assume that v has exactly two components, second one not 0.

Return: vector, representing rational function derivative of rational function (v[1]/v[2]) w.r.t. var(j)

Note: Possibly present non-commutative relations of the basering are ignored.

Example:

```
LIB "dmodloc.lib";
ring r = 0,(x,y),dp;
vector v = [x,y];
diffRat(v,1);
↦ y*gen(2)+gen(1)
diffRat(v,1) - [1,y];
↦ 0
diffRat(v,2);
↦ y2*gen(2)-x*gen(1)
diffRat(v,2) - [-x,y2];
↦ 0
```

7.7.22.24 commRing

Procedure from library `dmodloc.lib` (see Section 7.7.22 [dmodloc_lib], page 609).

Usage: commRing();

Return: ring, basering without non-commutative relations

Example:

```
LIB "dmodloc.lib";
def W = makeWeyl(3);
setring W; W;
↦ // coefficients: QQ
↦ // number of vars : 6
↦ //        block   1 : ordering dp
↦ //                  : names    x(1) x(2) x(3) D(1) D(2) D(3)
↦ //        block   2 : ordering C
↦ // noncommutative relations:
↦ //    D(1)x(1)=x(1)*D(1)+1
↦ //    D(2)x(2)=x(2)*D(2)+1
↦ //    D(3)x(3)=x(3)*D(3)+1
def W2 = commRing();
setring W2; W2;
↦ // coefficients: QQ
↦ // number of vars : 6
↦ //        block   1 : ordering dp
↦ //                  : names    x(1) x(2) x(3) D(1) D(2) D(3)
↦ //        block   2 : ordering C
ring r = 0,(x,y),dp;
def r2 = commRing(); // same as r
setring r2; r2;
↦ // coefficients: QQ
↦ // number of vars : 2
↦ //        block   1 : ordering dp
↦ //                  : names    x y
↦ //        block   2 : ordering C
```

7.7.22.25 rightNFWeyl

Procedure from library `dmodloc.lib` (see Section 7.7.22 [dmodloc_lib], page 609).

Usage: rightNFWeyl(id,k); id ideal or poly, k int

Assume: The basering is the n-th Weyl algebra over a field of characteristic 0 and for all 1<=i<=n the identity
var(i+n)*var(i)=var(i)*var(i+1)+1 holds, i.e. the sequence of variables is given by x(1),...,x(n),D(1),...,D(n), where D(i) is the differential operator belonging to x(i).

Return: same type as id, the right normal form of id with respect to the principal right ideal generated by the k-th variable

Note: No Groebner basis computation is used.

Example:

```
LIB "dmodloc.lib";
ring r = 0,(x,y,Dx,Dy),dp;
def W = Weyl();
setring W;
ideal I = x^3*Dx^3, y^2*Dy^2, x*Dy, y*Dx;
rightNFWeyl(I,1); // right NF wrt principal right ideal x*W
↦ _[1]=0
↦ _[2]=y^2*Dy^2
↦ _[3]=0
```

```
↦ _[4]=y*Dx
rightNFWeyl(I,3); // right NF wrt principal right ideal Dx*W
↦ _[1]=-6
↦ _[2]=y^2*Dy^2
↦ _[3]=x*Dy
↦ _[4]=0
rightNFWeyl(I,2); // right NF wrt principal right ideal y*W
↦ _[1]=x^3*Dx^3
↦ _[2]=0
↦ _[3]=x*Dy
↦ _[4]=0
rightNFWeyl(I,4); // right NF wrt principal right ideal Dy*W
↦ _[1]=x^3*Dx^3
↦ _[2]=2
↦ _[3]=0
↦ _[4]=y*Dx
poly p = x*Dx+1;
rightNFWeyl(p,1); // right NF wrt principal right ideal x*W
↦ 1
```

Appendix A Examples

A.1 Programming

A.1.1 Basic programming

We show in the example below the following:

- define the ring R of characteristic 32003, variables x,y,z, monomial ordering dp (implementing F_32003[x,y,z])
- list the information about R by typing its name
- check the order of the variables
- define the integers a,b,c,t
- define a polynomial f (depending on a,b,c,t) and display it
- define the jacobian ideal i of f
- compute a Groebner basis of i
- compute the dimension of the algebraic set defined by i (requires the computation of a Groebner basis)
- create and display a string in order to comment the result (text between quotes " "; is a 'string')
- load a library (see Section D.4.21 [primdec_lib], page 756)
- compute a primary decomposition for i and assign the result to a list L (which is a list of lists of ideals)
- display the number of primary components and the first primary and prime components (entries of the list L[1])
- implement the localization of F_32003[x,y,z] at the homogeneous maximal ideal (generated by x,y,z) by defining a ring with local monomial ordering (ds in place of dp)
- map i to this ring (see Section 5.1.57 [imap], page 196) - we may use the same name i, since ideals are ring dependent data
- compute the local dimension of the algebraic set defined by i at the origin (= dimension of the ideal generated by i in the localization)
- compute the local dimension of the algebraic set defined by i at the point (-2000,-6961,-7944) (by applying a linear coordinate transformation)

For a more basic introduction to programming in Singular, we refer to Section 2.3 [Getting started], page 6.

```
ring R = 32003,(x,y,z),dp;
R;
↦ // coefficients: ZZ/32003
↦ // number of vars : 3
↦ //        block   1 : ordering dp
↦ //                  : names    x y z
↦ //        block   2 : ordering C
x > y;
↦ 1
y > z;
↦ 1
```

```
int a,b,c,t = 1,2,-1,4;
poly f = a*x3+b*xy3-c*xz3+t*xy2z2;
f;
↦ 4xy2z2+2xy3+xz3+x3
ideal i = jacob(f);    // Jacobian Ideal of f
ideal si = std(i);     // compute Groebner basis
int dimi = dim(si);
string s = "The dimension of V(i) is "+string(dimi)+".";
s;
↦ The dimension of V(i) is 1.
LIB "primdec.lib";     // load library primdec.lib
list L = primdecGTZ(i);
size(L);               // number of prime components
↦ 6
L[1][1];               // first primary component
↦ _[1]=2y2z2+y3-16001z3
↦ _[2]=x
L[1][2];               // corresponding prime component
↦ _[1]=2y2z2+y3-16001z3
↦ _[2]=x
ring Rloc = 32003,(x,y,z),ds;   // ds = local monomial ordering
ideal i = imap(R,i);
dim(std(i));
↦ 1
map phi = R, x-2000, y-6961, z-7944;
dim(std(phi(i)));
↦ 0
```

A.1.2 Writing procedures and libraries

The user may add their own commands to the commands already available in Singular by writing Singular procedures. There are basically two kinds of procedures:

- procedures written in the Singular programming language (which are usually collected in Singular libraries).
- procedures written in C/C++ (collected in dynamic modules).

At this point, we restrict ourselves to describing the first kind of (library) procedures, which are sufficient for most applications. The syntax and general structure of a library (procedure) is described in Section 3.7 [Procedures], page 51, and Section 3.8 [Libraries], page 55.

The probably most efficient way of writing a new library is to use one of the official Singular libraries, say `ring.lib` as a sample. On a Unix-like operating system, type `LIB "ring.lib";` to get information on where the libraries are stored on your disk.

Singular provides several commands and tools, which may be useful when writing a procedure, for instance, to have a look at intermediate results (see Section 3.9 [Debugging tools], page 68).

If such a libarary should be contributed to Singular some formal requirements are needed:

the library header must explain the purpose of the library and (for non-trivial algorithm) a pointer to the algorithm (text book, article, etc.)

all global procedures must have a help string and an example which shows its usage.

it is strongly recommend also to provide test scripts which test the functionality: one should test the essential functionality of the library/command in a relatively short time (say, in no more than 30s), other tests should check the functionality of the library/command in detail so that, if possible, all relevant cases/results are tested. Nevertheless, such a test should not run longer than, say, 10 minutes.

We give short examples of procedures to demonstrate the following:

- Write procedures which return an integer (ring independent), see also Section A.4.1 [Milnor and Tjurina number], page 662. (Here we restrict ourselves to the main body of the procedures).
  - The procedure `milnorNumber` must be called with one parameter, a polynomial. The name g is local to the procedure and is killed automatically when leaving the procedure. `milnorNumber` returns the Milnor number (and displays a comment).
  - The procedure `tjurinaNumber` has no specified number of parameters. Here, the parameters are referred to by `#[1]` for the 1st, `#[2]` for the 2nd parameter, etc. `tjurinaNumber` returns the Tjurina number (and displays a comment).
  - the procedure `milnor_tjurina` which returns a list consisting of two integers, the Milnor and the Tjurina number.
- Write a procedure which creates a new ring and returns data dependent on this new ring (two numbers) and an int. In this example, we also show how to write a help text for the procedure (which is optional, but recommended).

```
proc milnorNumber (poly g)
{
   "Milnor number:";
   return(vdim(std(jacob(g))));
}

proc tjurinaNumber
{
   "Tjurina number:";
   return(vdim(std(jacob(#[1])+#[1])));
}

proc milnor_tjurina (poly f)
{
   ideal j=jacob(f);
   list L=vdim(std(j)),vdim(std(j+f));
   return(L);
}

proc real_sols (number b, number c)
"USAGE: real_sols (b,c);  b,c number
ASSUME: active basering has characteristic 0
RETURN: list: first entry is an integer (the number of different real
        solutions). If this number is non-negative, the list has as second
        entry a ring in which the list SOL of real solutions of x^2+bx+c=0
        is stored (as floating point number, precision 30 digits).
NOTE:   This procedure calls laguerre_solve from solve.lib.
"
{
  def oldring = basering;  // assign name to the ring active when
                           // calling the procedure
  number disc = b^2-4*c;
  if (disc>0) { int n_of_sols = 2; }
  if (disc==0) { int n_of_sols = 1; }
  string s = nameof(var(1));  // name of first ring variable
```

```
  if (disc>=0) {
    execute("ring rinC =(complex,30),("+s+"),lp;");
    if (not(defined(laguerre_solve))) { LIB "solve.lib"; }
    poly f = x2+imap(oldring,b)*x+imap(oldring,c);
                        // f is a local ring-dependent variable
    list SOL = laguerre_solve(f,30);
    export SOL;         // make SOL a global ring-dependent variable
                        // such variables are still accessible when the
                        // ring is among the return values of the proc
    setring oldring;
    return(list(n_of_sols,rinC));
  }
  else {
    return(list(0));
  }
}

//
// We now apply the procedures which are defined by the
// lines of code above:
//
ring r = 0,(x,y),ds;
poly f = x7+y7+(x-y)^2*x2y2;

milnorNumber(f);
↦ Milnor number:
↦ 28
tjurinaNumber(f);
↦ Tjurina number:
↦ 24
milnor_tjurina(f);     // a list containing Milnor and Tjurina number
↦ [1]:
↦    28
↦ [2]:
↦    24

def L=real_sols(2,1);
L[1];                  // number of real solutions of x^2+2x+1
↦ 1
def R1=L[2];
setring R1;
listvar(R1);           // only global ring-dependent objects are still alive
↦ // R1                           [0]  *ring
↦ // SOL                          [0]  list, size: 2
SOL;                   // the real solutions
↦ [1]:
↦ -1
↦ [2]:
↦ -1

setring r;
L=real_sols(1,1);
L[1];                  // number of reals solutions of x^2+x+1
```

```
↦ 0

setring r;
L=real_sols(1,-5);
L[1];                   // number of reals solutions of x^2+x-5
↦ 2
def R3=L[2];
setring R3; SOL;        // the real solutions
↦ [1]:
↦ -2.79128784747792000329402359686
↦ [2]:
↦ 1.79128784747792000329402359686
```

Writing a dynamic module is not as simple as writing a library procedure, since it does not only require some knowledge of C/C++, but also about the way the SINGULAR kernel works. See also Section A.1.9 [Dynamic modules], page 637.

A.1.3 Rings associated to monomial orderings

In SINGULAR we may implement localizations of the polynomial ring by choosing an appropriate monomial ordering (when defining the ring by the `ring` command). We refer to Section B.2 [Monomial orderings], page 696 for a thorough discussion of the monomial orderings available in SINGULAR.

At this point, we restrict ourselves to describing the relation between a monomial ordering and the ring (as mathematical object) which is implemented by the ordering. This is most easily done by describing the set of units: if `>` is a monomial ordering then precisely those elements which have leading monomial 1 are considered as units (in all computations performed with respect to this ordering).

In mathematical terms: choosing a monomial ordering `>` implements the localization of the polynomial ring with respect to the multiplicatively closed set of polynomials with leading monomial 1.

That is, choosing $>$ implements the ring

$$K[x]_> := \left\{ \frac{f}{u} \;\middle|\; f, u \in K[x],\, LM(u) = 1 \right\}.$$

If `>` is global (that is, 1 is the smallest monomial), the implemented ring is just the polynomial ring. If `>` is local (that is, if 1 is the largest monomial), the implemented ring is the localization of the polynomial ring w.r.t. the homogeneous maximal ideal. For a mixed ordering, we obtain "something in between these two rings":

```
ring R = 0,(x,y,z),dp;     // polynomial ring (global ordering)
poly f = y4z3+2x2y2z2+4z4+5y2+1;
f;                         // display f in a degrevlex-ordered way
↦ y4z3+2x2y2z2+4z4+5y2+1
short=0;                   // avoid short notation
f;
↦ y^4*z^3+2*x^2*y^2*z^2+4*z^4+5*y^2+1
short=1;
leadmonom(f);              // leading monomial
↦ y4z3

ring r = 0,(x,y,z),ds;     // local ring (local ordering)
```

```
poly f = fetch(R,f);
f;                        // terms of f sorted by degree
↦ 1+5y2+4z4+2x2y2z2+y4z3
leadmonom(f);             // leading monomial
↦ 1

// Now we implement more "advanced" examples of rings:
//
// 1)   (K[y]_<y>)[x]
//
int n,m=2,3;
ring A1 = 0,(x(1..n),y(1..m)),(dp(n),ds(m));
poly f  = x(1)*x(2)^2+1+y(1)^10+x(1)*y(2)^5+y(3);
leadmonom(f);
↦ x(1)*x(2)^2
leadmonom(1+y(1));        // unit
↦ 1
leadmonom(1+x(1));        // no unit
↦ x(1)

//
// 2)  some ring in between (K[x]_<x>)[y] and K[x,y]_<x>
//
ring A2 = 0,(x(1..n),y(1..m)),(ds(n),dp(m));
leadmonom(1+x(1));      // unit
↦ 1
leadmonom(1+x(1)*y(1));  // unit
↦ 1
leadmonom(1+y(1));      // no unit
↦ y(1)

//
// 3)  K[x,y]_<x>
//
ring A4 = (0,y(1..m)),(x(1..n)),ds;
leadmonom(1+y(1));      // in ground field
↦ 1
leadmonom(1+x(1)*y(1));  // unit
↦ 1
leadmonom(1+x(1));      // unit
↦ 1
```

Note, that even if we implictly compute over the localization of the polynomial ring, most computations are explicitly performed with polynomial data only. In particular, `1/(1-x);` does not return a power series expansion or a fraction but 0 (division with remainder in polynomial ring).

See Section 5.1.25 [division], page 171 for division with remainder in the localization and [invunit], page 782 for a procedure returning a truncated power series expansion of the inverse of a unit.

A.1.4 Long coefficients

The following innocent example produces in its standard basis extremely long coefficients in char 0 for the lexicographical ordering. But a very small deformation does not (the undeformed example is degenerated with respect to the Newton boundary). This example demonstrates that it might be wise, for complicated examples, to do the calculation first in positive char (e.g., 32003). It has been shown, that in complicated examples, more than 95 percent of the time needed for a standard basis computation is used in the computation of the coefficients (in char 0). The representation of long integers with real is demonstrated.

```
timer = 1;                             // activate the timer
ring R0 = 0,(x,y),lp;
poly f = x5+y11+xy9+x3y9;
ideal i = jacob(f);
ideal i1 = i,i[1]*i[2];                // undeformed ideal
ideal i2 = i,i[1]*i[2]+1/1000000*x5y8;  // deformation of i1
i1; i2;
↦ i1[1]=5x4+3x2y9+y9
↦ i1[2]=9x3y8+9xy8+11y10
↦ i1[3]=45x7y8+27x5y17+45x5y8+55x4y10+36x3y17+33x2y19+9xy17+11y19
↦ i2[1]=5x4+3x2y9+y9
↦ i2[2]=9x3y8+9xy8+11y10
↦ i2[3]=45x7y8+27x5y17+45000001/1000000x5y8+55x4y10+36x3y17+33x2y19+9xy17+1\
   1y19
ideal j = std(i1);
j;
↦ j[1]=264627y39+26244y35-1323135y30-131220y26+1715175y21+164025y17+1830125\
   y16
↦ j[2]=12103947791971846719838321886393392913750065060875xy8-28639152114168\
   3198701331939250003266767738632875y38-31954402206909026926764622877573565\
   78554430672591y37+57436621420822663849721381265738895282846320y36+1657764\
   214948799497573918210031067353932439400y35+213018481589308191195677223898\
   98682697001205500y34+1822194158663066565585991976961565719648069806148y33\
   -4701709279892816135156972313196394005220175y32-1351872269688192267600786\
   97600850686824231975y31-3873063305929810816961516976025038053001141375y30\
   +1325886675843874047990382005421144061861290080000y29+1597720195476063141\
   9467945895542406089526966887310y28-26270181336309092660633348002625330426\
   7126525y27-7586082690893335269027136248944859544727953125y26-867853074106\
   49464602285843351672148965395945625y25-5545808143273594102173252331151835\
   700278863924745y24+19075563013460437364679153779038394895638325y23+548562\
   322715501761058348996776922561074021125y22+157465452677648386073957464715\
   68100780933983125y21-1414279129721176222978654235817359505555191156250y20\
   -20711190069445893615213399650035715378169943423125y19+272942733337472665\
   573418092977905322984009750y18+789065115845334505801847294677413365720955\
   3750y17+63554897038491686787729656061044724651089803125y16-22099251729923\
   906699732244761028266074350255961625y14+147937139679655904353579489722585\
   91339027857296625y10
↦ j[3]=5x4+3x2y9+y9
// Compute average coefficient length (=51) by
//   - converting j[2] to a string in order to compute the number
//   of characters
//   - divide this by the number of monomials:
size(string(j[2])) div size(j[2]);
↦ 51
vdim(j);
↦ 63
// For a better representation normalize the long coefficients
```

```
// of the polynomial j[2] and map it to real:
poly p=(1/121039477919718467198383218863933929137500650608 75)*j[2];
ring R1=real,(x,y),lp;
short=0; // force the long output format
poly p=imap(R0,p);
p;
↦ x*y^8-(2.366e-02)*y^38-(2.640e-01)*y^37+(4.745e-06)*y^36+(1.370e-04)*y^35\
   +(1.760e-03)*y^34+(1.505e-01)*y^33-(3.884e-07)*y^32-(1.117e-05)*y^31-(3.2\
   00e-04)*y^30+(1.095e-01)*y^29+(1.320e+00)*y^28-(2.170e-05)*y^27-(6.267e-0\
   4)*y^26-(7.170e-03)*y^25-(4.582e-01)*y^24+(1.576e-06)*y^23+(4.532e-05)*y^\
   22+(1.301e-03)*y^21-(1.168e-01)*y^20-(1.711e+00)*y^19+(2.255e-05)*y^18+(6\
   .519e-04)*y^17+(5.251e-03)*y^16-(1.826e+00)*y^14+(1.222e+00)*y^10
// Compute a standard basis for the deformed ideal:
setring R0; // return to the original ring R0
j = std(i2);
j;
↦ j[1]=y16
↦ j[2]=65610xy8+17393508y27+7223337y23+545292y19+6442040y18-119790y14+80190\
   y10
↦ j[3]=5x4+3x2y9+y9
vdim(j);
↦ 40
```

A.1.5 Parameters

Let us deform the ideal in Section A.1.4 [Long coefficients], page 632 by introducing a parameter t and compute over the ground field Q(t). We compute the dimension at the generic point, i.e., $dim_{Q(t)} Q(t)[x,y]/j$. (This gives the same result as for the deformed ideal above. Hence, the above small deformation was "generic".)

For almost all $a \in Q$ this is the same as $dim_Q Q[x,y]/j_0$, where $j_0 = j|_{t=a}$.

```
  ring Rt = (0,t),(x,y),lp;
  Rt;
↦ // coefficients: QQ(t)
↦ // number of vars : 2
↦ //        block   1 : ordering lp
↦ //                  : names    x y
↦ //        block   2 : ordering C
  poly f = x5+y11+xy9+x3y9;
  ideal i = jacob(f);
  ideal j = i,i[1]*i[2]+t*x5y8;  // deformed ideal, parameter t
  vdim(std(j));
↦ 40
  ring R=0,(x,y),lp;
  ideal i=imap(Rt,i);
  int a=random(1,30000);
  ideal j=i,i[1]*i[2]+a*x5y8;  // deformed ideal, fixed integer a
  vdim(std(j));
↦ 40
```

A.1.6 Formatting output

We show how to insert the result of a computation inside a text by using strings. First we compute the powers of 2 and comment the result with some text. Then we do the same and give the output a nice format by computing and adding appropriate space.

```
  // The powers of 2:
  int  n;
  for (n = 2; n <= 128; n = n * 2)
  {"n = " + string (n);}
↦ n = 2
↦ n = 4
↦ n = 8
↦ n = 16
↦ n = 32
↦ n = 64
↦ n = 128
  // The powers of 2 in a nice format
  int j;
  string space = "";
  for (n = 2; n <= 128; n = n * 2)
  {
    space = "";
    for (j = 1; j <= 5 - size (string (n)); j = j+1)
    { space = space + " "; }
    "n =" + space + string (n);
  }
↦ n =    2
↦ n =    4
↦ n =    8
↦ n =   16
↦ n =   32
↦ n =   64
↦ n =  128
```

A.1.7 Cyclic roots

We write a procedure returning a string that enables us to create automatically the ideal of cyclic roots over the basering with n variables. The procedure assumes that the variables consist of a single letter each (hence no indexed variables are allowed; the procedure `cyclic` in `poly.lib` does not have this restriction). Then we compute a standard basis of this ideal and some numerical information. (This ideal is used as a classical benchmark for standard basis computations).

```
// We call the procedure 'cyclic':
proc cyclic (int n)
{
   string vs = varstr(basering)+varstr(basering);
   int c=find(vs,",");
   while ( c!=0 )
   {
      vs=vs[1,c-1]+vs[c+1,size(vs)];
      c=find(vs,",");
   }
   string t,s;
   int i,j;
```

```
   for ( j=1; j<=n-1; j=j+1 )
   {
      t="";
      for ( i=1; i <=n; i=i+1 )
      {
         t = t + vs[i,j] + "+";
      }
      t = t[1,size(t)-1] + ","+newline;
      s=s+t;
   }
   s=s+vs[1,n]+"-1";
   return (s);
}

ring r=0,(a,b,c,d,e),lp;         // basering, char 0, lex ordering
string sc=cyclic(nvars(basering));
sc;                              // the string of the ideal
↦ a+b+c+d+e,
↦ ab+bc+cd+de+ea,
↦ abc+bcd+cde+dea+eab,
↦ abcd+bcde+cdea+deab+eabc,
↦ abcde-1
execute("ideal i="+sc+";");      // this defines the ideal of cyclic roots
i;
↦ i[1]=a+b+c+d+e
↦ i[2]=ab+bc+cd+ae+de
↦ i[3]=abc+bcd+abe+ade+cde
↦ i[4]=abcd+abce+abde+acde+bcde
↦ i[5]=abcde-1
timer=1;
ideal j=std(i);
↦ //used time: 7.5 sec
size(j);                         // number of elements in the std basis
↦ 11
degree(j);
↦ // codimension = 5
↦ // dimension   = 0
↦ // degree      = 70
```

A.1.8 Parallelization with ssi links

In this example, we demonstrate how ssi links can be used to parallelize computations.

To compute a standard basis for a zero-dimensional ideal in the lexicographical ordering, one of the two powerful routines `stdhilb` (see [stdhilb], page 721) and `stdfglm` (see [stdfglm], page 721) should be used. However, in general one cannot predict which one of the two commands is faster. This very much depends on the (input) example. Therefore, we use ssi links to let both commands work on the problem independently and in parallel, so that the one which finishes first delivers the result.

The example we use is the so-called "omndi example". See *Tim Wichmann; Der FGLM-Algorithmus: verallgemeinert und implementiert in Singular; Diplomarbeit Fachbereich Mathematik, Universitaet Kaiserslautern; 1997* for more details.

```
ring r=0,(a,b,c,u,v,w,x,y,z),lp;
```

```
ideal i=a+c+v+2x-1, ab+cu+2vw+2xy+2xz-2/3,  ab2+cu2+2vw2+2xy2+2xz2-2/5,
ab3+cu3+2vw3+2xy3+2xz3-2/7, ab4+cu4+2vw4+2xy4+2xz4-2/9, vw2+2xyz-1/9,
vw4+2xy2z2-1/25, vw3+xyz2+xy2z-1/15, vw4+xyz3+xy3z-1/21;

link l_hilb,l_fglm = "ssi:fork","ssi:fork";         // 1.

open(l_fglm); open(l_hilb);

write(l_hilb, quote(stdhilb(i)));                       // 2.
write(l_fglm, quote(stdfglm(eval(i))));

list L=list(l_hilb,l_fglm);                             // 3.
int l_index=waitfirst(L);

if (l_index==1)
{
  "stdhilb won !!!!"; size(read(L[1]));
  close(L[1]); close(L[2]);
}
else                                                    // 4.
{
  "stdfglm won !!!!"; size(read(L[2]));
  close(L[1]); close(L[2]);
}
↦ stdfglm won !!!!
↦ 9
```

Some explanatory remarks are in order:

1. Instead of using links of the type `ssi:fork`, we alternatively could use `ssi:tcp` links such that the two "competing" Singular processes run on different machines. This has the advantage of "true" parallel computing since no resource sharing is involved (as it usually is with forked processes).
2. Notice how quoting is used in order to prevent local evaluation (i.e., local computation of results). Since we "forked" the two competing processes, the identifier `i` is defined and has identical values in both child processes. Therefore, the innermost `eval` can be omitted (as is done for the `l_hilb` link), and only the identifier `i` needs to be communicated to the children. However, when `ssi:tcp` links are used, the inner evaluation must be applied so that actual values, and not the identifiers are communicated (as is done for the `l_fglm` link in our example).
3. We wait until one of the two children finished the computation. The main process sleeps (i.e., suspends its execution) in the intermediate time.
4. The child which has won delivers the result and is terminated with the usual `close` command. The other child which is still computing needs to be terminated by an explicit (i.e., system) kill command if running on a different computer. For ssi:fork a `close` is sufficient.

A.1.9 Dynamic modules

The purpose of the following example is to illustrate the use of dynamic modules. Giving an example on how to write a dynamic module is beyond the scope of this manual. A technical reference is given at `http://www.singular.uni-kl.de/Manual/modules.pdf`.

In this example, we use a dynamic module, residing in the file `kstd.so`, which allows ignoring all but the first j entries of vectors when forming the pairs in the standard basis computation.

```
ring r=0,(x,y),dp;
module mo=[x^2-y^2,1,0,0],[xy+y^2,0,1,0],[y^2,0,0,1];
print(mo);

// load dynamic module - at the same time creating package Kstd
// procedures will be available in the packages Top and Kstd
LIB("kstd.so");
listvar(package);

// set the number of components to be considered to 1
module mostd=kstd(mo,1);        // calling procedure in Top
                    // obviously computation ignored pairs with leading
                    // term in the second entry
print(mostd);

// now consider 2 components
module mostd2=Kstd::kstd(mo,2); // calling procedure in Kstd
                    // this time the previously unconsidered pair was
                    // treated too
print(mostd2);
```

A.2 Computing Groebner and Standard Bases

A.2.1 groebner and std

The basic version of Buchberger's algorithm leaves a lot of freedom in carrying out the computational process. Considerable improvements are obtained by implementing criteria for reducing the number of S-polynomials to be actually considered (e.g., by applying the product criterion or the chain criterion). We refer to Cox, Little, and O'Shea [1997], Chapter 2 for more details and references on these criteria and on further strategies for improving the performance of Buchberger's algorithm (see also Greuel, Pfister [2002]).

Singular's implementation of Buchberger's algorithm is available via the `std` command ('std' referring to `st`an`d`ard basis). The computation of reduced Groebner and standard bases may be forced by setting `option(redSB)` (see Section 5.1.108 [option], page 230).

However, depending on the monomial ordering of the active basering, it may be advisable to use the `groebner` command instead. This command is provided by the Singular library `standard.lib` which is automatically loaded when starting a Singular session. Depending on some heuristics, `groebner` either refers to the `std` command (e.g., for rings with ordering `dp`), or to one of the algorithms described in the sections Section A.2.2 [Groebner basis conversion], page 640, Section A.2.3 [slim Groebner bases], page 642. For information on the heuristics behind `groebner`, see the library file `standard.lib` (see also Section 2.3.3 [Procedures and libraries], page 10).

We apply the commands `std` and `groebner` to compute a lexicographic Groebner basis for the ideal of cyclic roots over the basering with 6 variables (see Section A.1.7 [Cyclic roots], page 635). We set `option(prot)` to make Singular display some information on the performed computations (see Section 5.1.108 [option], page 230 for an interpretation of the displayed symbols). For long running computations, it is always recommended to set this option.

```
LIB "poly.lib";
ring r=32003,(a,b,c,d,e,f),lp;
ideal I=cyclic(6);
```

```
option(prot);
int t=timer;
system("--ticks-per-sec", 100);         // give time in 1/100 sec
ideal sI=std(I);
↦ [1023:1]1(5)s2(4)s3(3)s4s(4)s5(6)s(9)s(11)s(14)s(17)-s6s(19)s(21)s(24)s(2\
   7)s(30)s(33)s(35)s(38)s(41)ss(42)-s----s7(41)s(43)s(46)s(48)s(51)s(54)s(5\
   6)s(59)s(62)s(63)s(65)s(66)s(68)s(70)s(73)s(75)s(78)---ss(81)-----------s\
   (73)--------8-s(66)s(69)s(72)s(75)s(77)s(80)s(81)s(83)s(85)s(88)s(91)s(93\
   )s(96)s(99)s(102)s(105)s(107)s(110)s(113)-------------(100)-----------s(1\
   01)s(108)s(110)----------(100)--------9-s(94)s(97)s(99)s(84)s(74)s(77)s(8\
   0)---ss(83)s(86)s(73)s(76)s10(78)s(81)s(82)s(84)s(86)s(89)s(92)s(94)s(97)\
   s(100)s(103)s(82)s(84)s(86)s(89)s(92)s(95)s11(98)s(87)s(90)s(93)s(95)s(98\
   )s(101)s(104)----(100)---12-s(99)s(90)s(93)s(92)---------s(86)-----------\
   13-s(74)s(77)s(79)s(82)s(85)s(88)-----------------14-s(64)s(67)ss(70)s(7\
   3)s(77)s(81)-----------------------15-s(57)s(65)s(68)ss(71)--------------\
   ---s(57)----16-s(55)ss(56)------------------------17-s(34)s(32)--------18\
   -s(26)s(28)s-----19-s(25)s(28)s(31)------20-s(27)s(30)s(35)-------21-s(23\
   )s(26)------22-s(22)------23-s(15)24-s(17)-s(19)--25-s(18)s(19)s26-s(21)-\
   --------27-s(11)28-s(13)--29-s(12)-30--s--31-s(11)---32-s33(7)s(10)---34-\
   s-35----36-s37(6)s38s39s40---42-s43(5)s44s45--48-s49s50s51---54-s55(4)--6\
   7-86-
↦ product criterion:664 chain criterion:2844
timer-t;                                // used time (in 1/100 secs)
↦ 11
size(sI);
↦ 17
t=timer;
sI=groebner(I);
↦ compute hilbert series with std in ring (ZZ/32003),(a,b,c,d,e,f,@),(dp(7)\
   ,C)
↦ weights used for hilbert series: 1,1,1,1,1,1,1
↦ [511:1]1(5)s2(4)s3(3)s4ss5(4)s(5)s(7)-s6(8)s(9)s(11)s(13)s(16)s(18)s(21)-\
   -s7(22)s(23)s(24)s(27)s(29)s(31)s(32)s(35)-s(37)s(40)s(42)s(44)s(45)--s(4\
   6)s(48)-----8-s(44)s(47)s(50)s(52)s(55)s(57)s(59)s(61)-s(63)----s(62)----\
   s(61)s(64)-s(66)-----------s(58)-------9-s(53)s(56)s(59)s(62)s(65)s(68)s(\
   71)s(74)s(77)s(80)s(83)s(86)s(90)s(95)s(102)s(108)--------(100)----------\
   ------------s(81)---10-s(83)s(88)s(90)s(94)s(99)s(104)s(109)s(114)-s(116)\
   s(121)s(126)s(128)s(132)--------------------------------(100)------------\
   --11-s(87)---------------------------------------12-s(50)--------13-s(44)\
   s(47)s(51)s(55)-------------14-s(45)s(48)s(51)s(55)s(58)s(61)s(64)s(67)s(\
   70)--------------------15-s(52)s(55)s(58)s(61)s(64)s(67)s(70)s(73)s(76)s(\
   79)s(82)---------------------------------------16------------------------\
   ------------------17-
↦ product criterion:284 chain criterion:4184
↦ std with hilb in (ZZ/32003),(a,b,c,d,e,f,@),(lp(6),dp(1),C)
↦ [511:1]1(98)s2(97)s3(96)s4s(97)-s5(98)s(101)s(103)s(106)s(109)---s6(107)s\
   (109)s(111)s(114)s(117)s(120)s(123)s(125)s(128)s(131)ss(132)-s----------s\
   7(125)s(127)s(130)s(132)s(135)s(138)s(140)s(143)s(146)s(147)s(149)s(150)s\
   (152)s(154)s(157)s(159)s(162)---ss(165)-----------shhhhhhhhhhhhhhhhhhhhhh\
   h8(134)s(136)s(139)s(142)s(145)s(147)s(150)s(151)s(153)s(155)s(158)s(161)\
   s(163)s(166)s(169)s(172)s(175)s(177)s(180)s(183)------------------------s\
   (171)s(178)shhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhh9(147)s(150)s(153)s(155)s(18\
   1)s(184)s(187)s(190)s(203)s(208)s(213)s(217)s(218)s(220)s(222)s(225)---s-\
```

```
    s(226)-----------s(219)---------shhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhh\
    hhhhhhhhh10(163)s(166)s(168)s(171)s(177)s(180)s(183)s(186)shhhhhhhhhhhhhh\
    hhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhh11(125)s(128)s(130)\
    s(133)s(136)shhhhhhhhhhhhhhhhhhhhhhhhhhhhh12(110)s(113)s(120)s(123)s(127)\
    -------------shhhhhhhhhhhhhhh13(102)s(106)s(109)s(111)s(114)s(117)----shh\
    hhhhhhhhhhhhhhhhhhhhhhhhhhhhh14(85)s(90)s(93)s(97)s(100)s(103)---(100)-s(\
    103)shhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhh15(68)s(72)s(75)s(79)s(85)---\
    -shhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhh16(42)s(45)s(49)shhhhhhhhhhh\
    hhhhhhhhhhh17(34)s(37)shhhhhhhhhhhhhh18(27)s(30)s(32)-shhhhhhhhh19(26)s(2\
    9)s(32)shhhhhhhhhhhhh20(22)s(25)s(28)shhhhhhhhhhhhh21(20)s(26)shhhhhhhhhh\
    hh22(18)shhhhhhhhhh23(12)shhhhh24(11)s(14)-shhhh25(13)s(18)-s(21)shhhhhhh2\
    6(18)shhhhhhhhhhhhhh27(9)shhhhh28(8)shhhh29(7)shhhh30(8)-shhh31shhhhh32(7)s\
    hhhh33shhhh34(6)shhhhhhhh36(2)s37(6)shhhh38shhhh39shhhhhhhh42(2)s43(5)shh\
    hh44shhhhhhhh48s49shhhh50shhhhhhhh54shhhhh
↦ product criterion:720 chain criterion:11620
↦ hilbert series criterion:532
↦ dehomogenization
↦ simplification
↦ imap to ring (ZZ/32003),(a,b,c,d,e,f),(lp(6),C,L(1048575))
timer-t;                                // used time (in 1/100 secs)
↦ 4
size(sI);
↦ 17
option(noprot);
```

A.2.2 Groebner basis conversion

The performance of Buchberger's algorithm is sensitive to the chosen monomial order. A Groebner basis computation with respect to a less favorable order such as the lexicographic ordering may easily run out of time or memory even in cases where a Groebner basis computation with respect to a more efficient order such as the degree reverse lexicographic ordering is very well feasible. Groebner basis conversion algorithms and the Hilbert-driven Buchberger algorithm are based on this observation:

- Groebner basis conversion: Given an ideal $I \subset K[x_1, \ldots, x_n]$ and a slow monomial order, compute a Groebner basis with respect to an appropriately chosen fast order. Then convert the result to a Groebner basis with respect to the given slow order.
- Hilbert-driven Buchberger algorithm: Homogenize the given generators for I with respect to a new variable, say, x_0 . Extend the given slow ordering on $K[x_1, \ldots, x_n]$ to a global product ordering on $K[x_0, \ldots, x_n]$. Compute a Groebner basis for the ideal generated by the homogenized polynomials with respect to a fast ordering. Read the Hilbert function, and use this information when computing a Groebner basis with respect to the extended (slow) ordering. Finally, dehomogenize the elements of the resulting Groebner basis.

Singular provides implementations for the FGLM conversion algorithm (which applies to zero-dimensional ideals only, see [stdfglm], page 721) and variants of the Groebner walk conversion algorithm (which works for arbitrary ideals, See Section 5.1.47 [frwalk], page 186, Section D.4.7 [grwalk_lib], page 745). An implementation of the Hilbert-driven Buchberger algorithm is accessible via the `stdhilb` command (see also Section 5.1.147 [std], page 266).

For the ideal below, `stdfglm` is more than 100 times and `stdhilb` about 10 times faster than `std`.

```
ring r =32003,(a,b,c,d,e),lp;
ideal i=a+b+c+d, ab+bc+cd+ae+de, abc+bcd+abe+ade+cde,
        abc+abce+abde+acde+bcde, abcde-1;
```

```
  int t=timer;
  option(prot);
  ideal j1=stdfglm(i);
↦ std in (ZZ/32003),(a,b,c,d,e),(dp(5),C,L(1048575))
↦ [4095:2]1(4)s2(3)s3(2)s4s(3)s5s(4)s(5)s(6)6-ss(7)s(9)s(11)-7-ss(13)s(15)s\
  (17)--s--8-s(16)s(18)s(20)s(23)s(26)-s(23)-------9--s(16)s10(19)s(22)s(25\
  )----s(24)--s11---------s12(17)s(19)s(21)------s(17)s(19)s(21)s13(23)s--s\
  -----s(20)----------14-s(12)--------15-s(6)--16-s(5)--17---
↦ (S:21)---------------------
↦ product criterion:109 chain criterion:322
↦ .....+....-...-..-+-....-...-..---...-++---++---....-...-++---.++-----------...------....-...------+--------+---.++------++++-+++---------------+---
↦ vdim= 45
↦ ............................................++-------------------------------------------+--------------------------------------------+--------------------------------------------+--------------------------------------------
  timer-t;
↦ 0
  size(j1);   // size (no. of polys) in computed GB
↦ 5
  t=timer;
  ideal j2=stdhilb(i);
↦ compute hilbert series with std in ring (ZZ/32003),(a,b,c,d,e,@),(dp(6),C\
  ,L(1048575))
↦ weights used for hilbert series: 1,1,1,1,1,1
↦ [1023:1]1(4)s2(3)s3(2)s4ss5(3)s(4)s(5)-s6s(6)s(7)s(9)s(11)-7-ss(13)s(15)s\
  (17)--s--8-s(16)s(18)s(20)s(23)s(26)-s(29)-------9-s(25)s(28)--s(29)---s-\
  ------10-s(24)-------s(19)---11-s(17)s(19)s(21)-----s(18)-s(19)s12(21)s(2\
  3)s(26)-s(27)------s(23)----------13-s(15)-----------14-s(6)--15-s(5)--16\
  ---
↦ product criterion:88 chain criterion:650
↦ std with hilb in (ZZ/32003),(a,b,c,d,e,@),(lp(5),dp(1),C,L(1048575))
↦ [1023:1]1(41)s2(40)s3(39)s4s(40)-s5(41)s(44)s(46)s-s-sh6s(49)s(51)s(54)s(\
  55)s(56)s(58)s(59)--shhhhhhh7(53)s(55)s(57)s(59)s(61)-s(62)s(68)s(70)s(71\
  )s(74)--shhhhhhhhhhhhhhhh8(58)s(61)s(65)s(68)s(71)-s(72)s(75)--------shhh\
  hhhhhhhhhhhhhhhh9(51)s(53)s(56)s(58)s(61)s(64)------s(61)s(64)shhhhhhhhhh\
  hhhhh10(53)s(55)s(58)s(62)s(64)s(67)s(70)--s(71)------s(68)s(71)s(73)--sh\
  hhhhhhhhhhhhhhh11(58)s(60)s(63)s(66)s(69)s(72)s(74)---s-s(76)s(79)----s(7\
  8)-------shhhhhhhhhhhhhhhhhhhhhhh12(51)s(54)s(57)s(58)s(60)s(63)s(65)s(68\
  )s(70)s(73)s(76)s(79)--s(80)----shhhhhhhhhhhhhhhhhhhhhhhhhhhhhh13(48)s(51\
  )s(54)s(57)s(59)s(61)s(64)s(67)shhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhh14\
  (31)s(33)s(36)s(39)s(42)s(45)shhhhhhhhhhhhhhhhhhhhhhhhh15(23)s(26)s(29)s(\
  32)s(35)shhhhhhhhhhhhhhhhhhh16(18)s(21)s(24)s(27)shhhhhhhhhhhhhhh17(15)s(\
  18)s(21)s(24)shhhhhhhhhhhh18(15)s(18)s(21)s(24)shhhhhhhhhhhh19(14)s(17)s(\
  20)shhhhhhhhhhhh20(11)s(14)s(17)shhhhhhhhh21(11)s(14)s(17)shhhhhhhhh22(11\
  )s(14)s(16)shhhhhhhhh23(10)s(13)shhhhhhhhh24(7)s(10)shhhhhh25(7)s(10)shhh\
  hhh26(7)s(10)shhhhhh27(7)s(10)shhhhhh28(7)s(10)shhhhhh29(7)s(10)shhhhhh30\
  (7)s(9)shhhhhh31(6)shhhhhh32(3)shhh33shhh34shhh35shhh36shhh37shhh38shhh39\
  shhh40shhh41shhh42shhh43shhh44shhh45shhh46shhh47shhh48shhh49shhh50shhh51s\
  hhh52shhh53shhh54shhhhhh
↦ product criterion:491 chain criterion:11799
↦ hilbert series criterion:417
↦ dehomogenization
↦ simplification
↦ imap to ring (ZZ/32003),(a,b,c,d,e),(lp(5),C,L(1048575))
  timer-t;
```

```
↦ 0
  size(j2);   // size (no. of polys) in computed GB
↦ 5
  // usual Groebner basis computation for lex ordering
  t=timer;
  ideal j0 =std(i);
↦ [4095:1]1(4)s2(3)s3(2)s4s(3)s5(5)s(4)s6(6)s(7)s(9)s(8)sss7(10)s(11)s(10)s\
   (11)s(13)s8(12)s(13)s(15)s.s(14).s.9.s(16)s(17)s(19)........10.s(20).s(21\
   )ss..11.s(23)s(25).ss(27)...s(28)s(26)...12.s(25)sss(23)sss......s(22)..\
   .13.s(23)ssssssss(21)s(22)sssss(21)ss..14.ss(22)s.s.sssss(21)s(22)sss.s..\
   .15.ssss(21)s(22)ssssssssss(21)s(22)sss16.ssssssss(21)s(22)sssssssssss17s\
   s(21)s(22)ssssssssss(21)sss(22)ss(21)ss18(22)s(21)s(22)s.s..............1\
   9.sssss(21)ss(22)ssssssssss(21)s(22)s20.ssssssssss(21)s........21.s(22)ss\
   ssssssssssss(21)s(22)ssss22ssssssssssss(21)s(22)sssssss23sssssssssss(21)\
   s(22)ssssssss24ssssssssssss(21)s(22)sssssss25ssssssssss(21)s(22)sssssssss\
   26ssssssssss(21)s(20)ssssssss27.sssssssss..........s28.ssssss............\
   .29.sssssssssssssssssss30sssssssssssssssssss31.ssssssssssssssssss32.ssss\
   ssssssssssssssss33sssssssssssssssssss34sssssssssssssssssss35ssssssssssss\
   sssssssss36sssssssssssssssssss37sssssssssssssssssss38ssssssssssssssssss\
   ss39sssssssssssssssssss40sssssssssssssssssss41ssss---------------42-s(4\
   )--43-s44s45s46s47s48s49s50s51s52s53s54s55s56s
↦ product criterion:1395 chain criterion:904
  option(noprot);
  timer-t;
↦ 0
```

A.2.3 slim Groebner bases

The command `slimgb` calls an implementation of an algorithm to compute Groebner bases which is designed for keeping the polynomials slim (short with small coefficients) during a Groebner basis computation. It provides, in particular, a fast algorithm for computing Groebner bases over function fields or over the rational numbers, but also in several other cases. The algorithm which is still under developement was developed in the diploma thesis of Michael Brickenstein. It has been published as `http://www.mathematik.uni-kl.de/~zca/Reports_on_ca/35/paper_35_full.ps.gz`.

In the example below (Groebner basis with respect to degree reverse lexicographic ordering over function field) `slimgb` is much faster than the `std` command.

```
ring r=(32003,u1, u2, u3, u4),(x1, x2, x3, x4, x5, x6, x7),dp;
timer=1;
ideal i=
-x4*u3+x5*u2,
x1*u3+2*x2*u1-2*x2*u2-2*x3*u3-u1*u4+u2*u4,
-2*x1*x5+4*x4*x6+4*x5*x7+x1*u3-2*x4*u1-2*x4*u4-2*x6*u2-2*x7*u3+u1*u2+u2*u4,
-x1*x5+x1*x7-x4*u1+x4*u2-x4*u4+x5*u3+x6*u1-x6*u2+x6*u4-x7*u3,
-x1*x4+x1*u1-x5*u1+x5*u4,
-2*x1*x3+x1*u3-2*x2*u4+u1*u4+u2*u4,
x1^2*u3+x1*u1*u2-x1*u2^2-x1*u3^2-u1*u3*u4+u3*u4^2;
i=slimgb(i);
```

For detailed information and limitations see Section 5.1.141 [slimgb], page 261.

A.3 Commutative Algebra

A.3.1 Saturation

For any two ideals i, j in the basering R let

$$\mathrm{sat}(i,j) = \{x \in R \mid \exists\, n \text{ s.t. } x \cdot (j^n) \subseteq i\} = \bigcup_{n=1}^{\infty} i : j^n$$

denote the saturation of i with respect to j . This defines, geometrically, the closure of the complement of V(j) in V(i) (where V(i) denotes the variety defined by i).

The saturation is computed by the procedure `sat` in `elim.lib` by computing iterated ideal quotients with the maximal ideal. `sat` returns a list of two elements: the saturated ideal and the number of iterations.

We apply saturation to show that a variety has no singular points outside the origin (see also Section A.4.2 [Critical points], page 664). We choose m to be the homogeneous maximal ideal (note that `maxideal(n)` denotes the n-th power of the maximal ideal). Then $V(i)$ has no singular point outside the origin if and only if $sat(j+(f), m)$ is the whole ring, that is, generated by 1.

```
LIB "elim.lib";         // loading library elim.lib
ring r2 = 32003,(x,y,z),dp;
poly f = x^11+y^5+z^(3*3)+x^(3+2)*y^(3-1)+x^(3-1)*y^(3-1)*z3+
  x^(3-2)*y^3*(y^2)^2;
ideal j=jacob(f);
sat(j+f,maxideal(1));
↦ [1]:
↦    _[1]=1
↦ [2]:
↦    17
// list the variables defined so far:
listvar();
↦ // r2                          [0]  *ring
↦ //      j                           [0]  ideal, 3 generator(s)
↦ //      f                           [0]  poly
```

A.3.2 Finite fields

We define a variety in the n -space of codimension 2 defined by polynomials of degree d with generic coefficients over the prime field Z/p and look for zeros on the torus. First over the prime field and then in the finite extension field with p^k elements. In general there will be many more solutions in the second case. (Since the SINGULAR language is interpreted, the evaluation of many `for`-loops is not very fast):

```
int p=3;  int n=3;  int d=5; int k=2;
ring rp = p,(x(1..n)),dp;
int s = size(maxideal(d));
s;
↦ 21
// create a dense homogeneous ideal m, all generators of degree d, with
// generic (random) coefficients:
ideal m = maxideal(d)*random(p,s,n-2);
m;
↦ m[1]=x(1)^3*x(2)^2-x(1)*x(2)^4+x(1)^4*x(3)-x(1)^3*x(2)*x(3)+x(1)*x(2)^3*x\
  (3)+x(2)^4*x(3)+x(2)^3*x(3)^2+x(1)*x(2)*x(3)^3+x(1)*x(3)^4-x(3)^5
// look for zeros on the torus by checking all points (with no component 0)
```

```
  // of the affine n-space over the field with p elements :
  ideal mt;
  int i(1..n);                     // initialize integers i(1),...,i(n)
  int l;
  s=0;
  for (i(1)=1;i(1)<p;i(1)=i(1)+1)
  {
    for (i(2)=1;i(2)<p;i(2)=i(2)+1)
    {
      for (i(3)=1;i(3)<p;i(3)=i(3)+1)
      {
        mt=m;
        for (l=1;l<=n;l=l+1)
        {
          mt=subst(mt,x(l),i(l));
        }
        if (size(mt)==0)
        {
          "solution:",i(1..n);
          s=s+1;
        }
      }
    }
  }
↦ solution: 1 1 2
↦ solution: 1 2 1
↦ solution: 1 2 2
↦ solution: 2 1 1
↦ solution: 2 1 2
↦ solution: 2 2 1
  "//",s,"solutions over GF("+string(p)+")";
↦ // 6 solutions over GF(3)
  // Now go to the field with p^3 elements:
  // As long as there is no map from Z/p to the field with p^3 elements
  // implemented, use the following trick: convert the ideal to be mapped
  // to the new ring to a string and then execute this string in the
  // new ring
  string ms="ideal m="+string(m)+";";
  ms;
↦ ideal m=x(1)^3*x(2)^2-x(1)*x(2)^4+x(1)^4*x(3)-x(1)^3*x(2)*x(3)+x(1)*x(2)^\
   3*x(3)+x(2)^4*x(3)+x(2)^3*x(3)^2+x(1)*x(2)*x(3)^3+x(1)*x(3)^4-x(3)^5;
  // define a ring rpk with p^k elements, call the primitive element z. Hence
  // 'solution exponent: 0 1 5' means that (z^0,z^1,z^5) is a solution
  ring rpk=(p^k,z),(x(1..n)),dp;
  rpk;
↦ // coefficients: ZZ/9[z]
↦ //   minpoly        : 1*z^2+2*z^1+2*z^0
↦ // number of vars : 3
↦ //        block   1 : ordering dp
↦ //                  : names    x(1) x(2) x(3)
↦ //        block   2 : ordering C
  execute(ms);
  s=0;
```

```
  ideal mt;
  for (i(1)=0;i(1)<p^k-1;i(1)=i(1)+1)
  {
    for (i(2)=0;i(2)<p^k-1;i(2)=i(2)+1)
    {
      for (i(3)=0;i(3)<p^k-1;i(3)=i(3)+1)
      {
        mt=m;
        for (l=1;l<=n;l=l+1)
        {
          mt=subst(mt,x(l),z^i(l));
        }
        if (size(mt)==0)
        {
          // we show only the first 7 solutions here:
          if (s<5) {"solution exponent:",i(1..n);}
          s=s+1;
        }
      }
    }
  }
↦ solution exponent: 0 0 2
↦ solution exponent: 0 0 4
↦ solution exponent: 0 0 6
↦ solution exponent: 0 4 0
↦ solution exponent: 0 4 1
  "//",s,"solutions over GF("+string(p^k)+")";
↦ // 72 solutions over GF(9)
```

A.3.3 Elimination

Elimination is the algebraic counterpart of the geometric concept of projection. If $f = (f_1, \ldots, f_n) : k^r \to k^n$ is a polynomial map, the Zariski-closure of the image is the zero-set of the ideal $j = J \cap k[x_1, \ldots, x_n]$, where

$$J = (x_1 - f_1(t_1, \ldots, t_r), \ldots, x_n - f_n(t_1, \ldots, t_r)) \subseteq k[t_1, \ldots, t_r, x_1, \ldots, x_n]$$

that is, of the ideal j obtained from J by eliminating the variables $t_1, \ldots, t_r$. This can be done by computing a Groebner basis for J with respect to a (global) product ordering where the block of t-variables precedes the block of x-variables, and then selecting those polynomials which do not contain any t. Alternatively, we may use a global monomial ordering with extra weight vector (see Section B.2.8 [Extra weight vector], page 700), assigning to the t-variables a positive weight and to the x-variables weight 0.

Since elimination is expensive, it may be useful to use a Hilbert-driven approach to the elimination problem (see Section A.2.2 [Groebner basis conversion], page 640):

First compute the Hilbert function of the ideal w.r.t. a fast ordering (e.g., `dp`), then make use of it to speed up the computation by a Hilbert-driven elimination which uses the `intvec` provided as third argument.

In SINGULAR the most convenient way is to use the `eliminate` command. In contrast to the first method, with `eliminate` the result needs not be a standard basis in the given ordering. Hence, there may be cases where the first method is the preferred one.

WARNING: In the case of a local or a mixed ordering, elimination needs special care. f may be considered as a map of germs $f : (k^r, 0) \to (k^n, 0)$, but even if this map germ is finite, we are in general not able to compute the image germ because for this we would need an implementation of the Weierstrass preparation theorem. What we can compute, and what `eliminate` actually does, is the following: let V(J) be the zero-set of J in $k^r \times (k^n, 0)$, then the closure of the image of V(J) under the projection

$$\mathrm{pr} : k^r \times (k^n, 0) \to (k^n, 0)$$

can be computed. (Note that this germ contains also those components of V(J) which meet the fiber of pr outside the origin.) This is achieved by an ordering with the block of t-variables having a global ordering (and preceding the x-variables) and the x-variables having a local ordering.

In any case, if the input is weighted homogeneous (=quasihomogeneous), the weights given to the variables should be chosen accordingly. SINGULAR offers a function `weight` which proposes, given an ideal or module, integer weights for the variables, such that the ideal, resp. module, is as homogeneous as possible with respect to these weights. The function finds correct weights, if the input is weighted homogeneous (but is rather slow for many variables). In order to check, whether the input is quasihomogeneous, use the function `qhweight`, which returns an `intvec` of correct weights if the input is quasihomogeneous and an `intvec` of zeros otherwise.

Let us give three examples:

1. First we compute the equations of the simple space curve 'T[7]' consisting of two tangential cusps given in parametric form.
2. We compute weights for the equations such that the equations are quasihomogeneous w.r.t. these weights.
3. Then we compute the tangent developable of the rational normal curve in P^4.

```
  // 1. Compute equations of curve given in parametric form:
  // Two transversal cusps in (k^3,0):
  ring r1 = 0,(t,x,y,z),ls;
  ideal i1 = x-t2,y-t3,z;        // parametrization of the first branch
  ideal i2 = y-t2,z-t3,x;        // parametrization of the second branch
  ideal j1 = eliminate(i1,t);
  j1;                            // equations of the first branch
↦ j1[1]=z
↦ j1[2]=y2-x3
  ideal j2 = eliminate(i2,t);
  j2;                            // equations of the second branch
↦ j2[1]=z2-y3
↦ j2[2]=x
  // Now map to a ring with only x,y,z as variables and compute the
  // intersection of j1 and j2 there:
  ring r2 = 0,(x,y,z),ds;
  ideal j1= imap(r1,j1);         // imap is a convenient ringmap for
  ideal j2= imap(r1,j2);         // inclusions and projections of rings
  ideal i = intersect(j1,j2);
  i;                             // equations of both branches
↦ i[1]=z2-y3+x3y
↦ i[2]=xz
↦ i[3]=xy2-x4
  //
  // 2. Compute the weights:
  intvec v= qhweight(i);         // compute weights
  v;
```

```
↦ 4,6,9
  //
  // 3. Compute the tangent developable
  // The tangent developable of a projective variety given parametrically
  // by F=(f1,...,fn) : P^r --> P^n is the union of all tangent spaces
  // of the image. The tangent space at a smooth point F(t1,...,tr)
  // is given as the image of the tangent space at (t1,...,tr) under
  // the tangent map (affine coordinates)
  //   T(t1,...,tr): (y1,...,yr) --> jacob(f)*transpose((y1,...,yr))
  // where jacob(f) denotes the jacobian matrix of f with respect to the
  // t's evaluated at the point (t1,...,tr).
  // Hence we have to create the graph of this map and then to eliminate
  // the t's and y's.
  // The rational normal curve in P^4 is given as the image of
  //        F(s,t) = (s4,s3t,s2t2,st3,t4)
  // each component being homogeneous of degree 4.
  ring P = 0,(s,t,x,y,a,b,c,d,e),dp;
  ideal M = maxideal(1);
  ideal F = M[1..2];     // take the 1st two generators of M
  F=F^4;
  // simplify(...,2); deletes 0-columns
  matrix jac = simplify(jacob(F),2);
  ideal T = x,y;
  ideal J = jac*transpose(T);
  ideal H = M[5..9];
  ideal i = matrix(H)-matrix(J);// this is tricky: difference between two
                       // ideals is not defined, but between two
                       // matrices. By type conversion
                       // the ideals are converted to matrices,
                       // subtracted and afterwards converted
                       // to an ideal. Note that '+' is defined
                       // and adds (concatenates) two ideals
  i;
↦ i[1]=-4s3x+a
↦ i[2]=-3s2tx-s3y+b
↦ i[3]=-2st2x-2s2ty+c
↦ i[4]=-t3x-3st2y+d
↦ i[5]=-4t3y+e
  // Now we define a ring with product ordering and weights 4
  // for the variables a,...,e.
  // Then we map i from P to P1 and eliminate s,t,x,y from i.
  ring P1 = 0,(s,t,x,y,a,b,c,d,e),(dp(4),wp(4,4,4,4,4));
  ideal i = fetch(P,i);
  ideal j= eliminate(i,stxy);    // equations of tangent developable
  j;
↦ j[1]=3c2-4bd+ae
↦ j[2]=2bcd-3ad2-3b2e+4ace
↦ j[3]=8b2d2-9acd2-9b2ce+14abde-4a2e2
  // We can use the product ordering to eliminate s,t,x,y from i
  // by a std-basis computation.
  // We need proc 'nselect' from elim.lib.
  LIB "elim.lib";
  j = std(i);                    // compute a std basis j
```

```
  j = nselect(j,1..4);          // select generators from j not
  j;                            // containing variable 1,...,4
↦ j[1]=3c2-4bd+ae
↦ j[2]=2bcd-3ad2-3b2e+4ace
↦ j[3]=8b2d2-9acd2-9b2ce+12ac2e-2abde
```

A.3.4 Free resolution

In SINGULAR a free resolution of a module or ideal has its own type: `resolution`. It is a structure that stores all information related to free resolutions. This allows partial computations of resolutions via the command `res`. After applying `res`, only a pre-format of the resolution is computed which allows to determine invariants like Betti-numbers or homological dimension. To see the differentials of the complex, a resolution must be converted into the type list which yields a list of modules: the k-th module in this list is the first syzygy-module (module of relations) of the (k-1)st module. There are the following commands to compute a resolution:

`res` [res], page 721
computes a free resolution of an ideal or module using a heuristically chosen method. This is the preferred method to compute free resolutions of ideals or modules.

`fres` Section 5.1.46 [fres], page 185
improved version of Section 5.1.145 [sres], page 264, computes a free resolution of an ideal or module using Schreyer's method. The input has to be a standard basis.

`lres` Section 5.1.81 [lres], page 212
computes a free resolution of an ideal or module with LaScala's method. The input needs to be homogeneous.

`mres` Section 5.1.96 [mres], page 223
computes a minimal free resolution of an ideal or module with the syzygy method.

`sres` Section 5.1.145 [sres], page 264
computes a free resolution of an ideal or module with Schreyer's method. The input has to be a standard basis.

`nres` Section 5.1.103 [nres], page 228
computes a free resolution of an ideal or module with the standard basis method.

`minres` Section 5.1.91 [minres], page 220
minimizes a free resolution of an ideal or module.

`syz` Section 5.1.152 [syz], page 275
computes the first syzygy module.

`res(i,r)`, `lres(i,r)`, `sres(i,r)`, `mres(i,r)`, `nres(i,r)` compute the first r modules of the resolution of i, resp. the full resolution if r=0 and the basering is not a qring. See the manual for a precise description of these commands.
Note: The command `betti` does not require a minimal resolution for the minimal Betti numbers.

Now let us take a look at an example which uses resolutions: The Hilbert-Burch theorem says that the ideal i of a reduced curve in K^3 has a free resolution of length 2 and that i is given by the 2x2 minors of the 2nd matrix in the resolution. We test this for two transversal cusps in K^3. Afterwards we compute the resolution of the ideal j of the tangent developable of the rational normal curve in P^4 from above. Finally we demonstrate the use of the type `resolution` in connection with the `lres` command.

```
  // Two transversal cusps in (k^3,0):
  ring r2 =0,(x,y,z),ds;
  ideal i =z2-1y3+x3y,xz,-1xy2+x4,x3z;
  resolution rs=mres(i,0);   // computes a minimal resolution
  rs;                        // the standard representation of complexes
↦   1       3       2
↦ r2 <--  r2 <--  r2
↦
↦ 0       1       2
↦
    list resi=rs;            // convertion to a list
  print(resi[1]);            // the 1st module is i minimized
↦ xz,
↦ z2-y3+x3y,
↦ xy2-x4
  print(resi[2]);            // the 1st syzygy module of i
↦ -z,-y2+x3,
↦ x, 0,
↦ y, z
  resi[3];                   // the 2nd syzygy module of i
↦ _[1]=0
  ideal j=minor(resi[2],2);
  reduce(j,std(i));          // check whether j is contained in i
↦ _[1]=0
↦ _[2]=0
↦ _[3]=0
  size(reduce(i,std(j)));    // check whether i is contained in j
↦ 0
  // size(<ideal>) counts the non-zero generators
  // ---------------------------------------------
  // The tangent developable of the rational normal curve in P^4:
  ring P = 0,(a,b,c,d,e),dp;
  ideal j= 3c2-4bd+ae, -2bcd+3ad2+3b2e-4ace,
           8b2d2-9acd2-9b2ce+9ac2e+2abde-1a2e2;
  resolution rs=mres(j,0);
  rs;
↦  1      2      1
↦ P <--  P <--  P
↦
↦ 0      1      2
↦
  list L=rs;
  print(L[2]);
↦ 2bcd-3ad2-3b2e+4ace,
↦ -3c2+4bd-ae
  // create an intmat with graded Betti numbers
  intmat B=betti(rs);
  // this gives a nice output of Betti numbers
  print(B,"betti");
↦            0     1     2
↦ ------------------------
↦     0:     1     -     -
↦     1:     -     1     -
```

```
↦      2:     -     1     -
↦      3:     -     -     1
↦ ------------------------
↦ total:     1     2     1
↦
  // the user has access to all Betti numbers
  // the 2-nd column of B:
  B[1..4,2];
↦ 0 1 1 0
  ring cyc5=32003,(a,b,c,d,e,h),dp;
  ideal i=
  a+b+c+d+e,
  ab+bc+cd+de+ea,
  abc+bcd+cde+dea+eab,
  abcd+bcde+cdea+deab+eabc,
  h5-abcde;
  resolution rs=lres(i,0);   //computes the resolution according LaScala
  rs;                        //the shape of the minimal resolution
↦     1         5         10         10         5         1
↦ cyc5 <--  cyc5 <--  cyc5 <--   cyc5 <--   cyc5 <--  cyc5
↦
↦ 0         1         2          3          4         5
↦
  print(betti(rs),"betti");  //shows the Betti-numbers of cyclic 5
↦            0     1     2     3     4     5
↦ ------------------------------------------
↦      0:     1     1     -     -     -     -
↦      1:     -     1     1     -     -     -
↦      2:     -     1     1     -     -     -
↦      3:     -     1     2     1     -     -
↦      4:     -     1     2     1     -     -
↦      5:     -     -     2     2     -     -
↦      6:     -     -     1     2     1     -
↦      7:     -     -     1     2     1     -
↦      8:     -     -     -     1     1     -
↦      9:     -     -     -     1     1     -
↦     10:     -     -     -     -     1     1
↦ ------------------------------------------
↦ total:     1     5    10    10     5     1
↦
  dim(rs);                   //the homological dimension
↦ 4
  size(list(rs));            //gets the full (non-reduced) resolution
↦ 6
  minres(rs);                //minimizes the resolution
↦     1         5         10         10         5         1
↦ cyc5 <--  cyc5 <--  cyc5 <--   cyc5 <--   cyc5 <--  cyc5
↦
↦ 0         1         2          3          4         5
↦
  size(list(rs));            //gets the minimized resolution
↦ 6
```

A.3.5 Handling graded modules

How to deal with graded modules in Singular is best explained by looking at an example:

```
ring R = 0, (w,x,y,z), dp;
module I = [-x,0,-z2,0,y2z], [0,-x,-yz,0,y3], [-w,0,0,yz,-z3],
           [0,-w,0,y2,-yz2], [0,-1,-w,0,xz], [0,-w,0,y2,-yz2],
           [x2,-y2,-wy2+xz2];
print(I);
↦ -x, 0,  -w, 0,   0, 0,   x2,
↦ 0,  -x, 0,  -w,  -1,-w,  -y2,
↦ -z2,-yz,0,  0,   -w,0,   -wy2+xz2,
↦ 0,  0,  yz, y2,  0, y2,  0,
↦ y2z,y3, -z3,-yz2,xz,-yz2,0

// (1) Check on degrees:
// =====================
attrib(I,"isHomog"); // attribute not set => empty output
↦
homog(I);
↦ 1
attrib(I,"isHomog");
↦ 2,2,1,1,0

print(betti(I,0),"betti");  // read degrees from Betti diagram
↦            0     1
↦ ------------------
↦     0:     1     -
↦     1:     2     1
↦     2:     2     5
↦     3:     -     1
↦ ------------------
↦ total:     5     7
↦

// (2) Shift degrees:
// ==================
def J=I;
intvec DV = 0,0,-1,-1,-2;
attrib(J,"isHomog",DV);    // assign new weight vector
attrib(J,"isHomog");
↦ 0,0,-1,-1,-2
print(betti(J,0),"betti");
↦            0     1
↦ ------------------
↦    -2:     1     -
↦    -1:     2     1
↦     0:     2     5
↦     1:     -     1
↦ ------------------
↦ total:     5     7
↦

intmat bettiI=betti(I,0);  // degree corresponding to first non-zero row
```

```
                                // of Betti diagram is accessible via
                                // attribute "rowShift"
attrib(bettiI);
↦ attr:rowShift, type int
intmat bettiJ=betti(J,0);
attrib(bettiJ);
↦ attr:rowShift, type int

// (3) Graded free resolutions:
// ===========================
resolution resJ = mres(J,0);
attrib(resJ);
↦ attr:isHomog, type intvec
print(betti(resJ),"betti");
↦            0     1     2
↦ ------------------------
↦    -2:     1     -     -
↦    -1:     2     -     -
↦     0:     1     4     -
↦     1:     -     -     1
↦ ------------------------
↦ total:     4     4     1
↦
attrib(betti(resJ));
↦ attr:rowShift, type int
```

A check on degrees ((1), by using the `homog` command) shows that this is a graded matrix. The `homog` command assigns an admissible weight vector (here: 2,2,1,1,0) to the module `I` which is accessible via the attribute `"isHomog"`. Thus, we may think of `I` as a graded submodule of the graded free R -module

$$F = R(-2)^2 \oplus R(-1)^2 \oplus R\,.$$

We may also read the degrees from the Betti diagram as shown above. The degree on the left of the first nonzero row of the Betti diagram is accessible via the attribute `"rowShift"` of the betti matrix (which is of type `intmat`):

(2) We may shift degrees by assigning another admissible degree vector. Note that SINGULAR does not check whether the assigned degree vector really is admissible. Moreover, note that all assigned attributes are lost under a type conversion (e.g. from `module` to `matrix`).

(3) These considerations may be applied when computing data from free resolutions (see Section A.3.6 [Computation of Ext], page 652).

A.3.6 Computation of Ext

We start by showing how to calculate the n-th Ext group of an ideal. The ingredients to do this are by the definition of Ext the following: calculate a (minimal) resolution at least up to length n, apply the Hom functor, and calculate the n-th homology group, that is, form the quotient ker/im in the resolution sequence.

The Hom functor is given simply by transposing (hence dualizing) the module or the corresponding matrix with the command `transpose`. The image of the (n-1)-st map is generated by the columns of the corresponding matrix. To calculate the kernel apply the command `syz` at the (n-1)-st transposed entry of the resolution. Finally, the quotient is obtained by the command `modulo`, which gives for two modules A = ker, B = Im the module of relations of

$$A/(A \cap B)$$

in the usual way. As we have a chain complex, this is obviously the same as ker/Im.

We collect these statements in the following short procedure:

```
proc ext(int n, ideal I)
{
  resolution rs = mres(I,n+1);
  module tAn    = transpose(rs[n+1]);
  module tAn_1  = transpose(rs[n]);
  module ext_n  = modulo(syz(tAn),tAn_1);
  return(ext_n);
}
```

Now consider the following example:

```
ring r5 = 32003,(a,b,c,d,e),dp;
ideal I = a2b2+ab2c+b2cd, a2c2+ac2d+c2de,a2d2+ad2e+bd2e,a2e2+abe2+bce2;
print(ext(2,I));
↦ 1,0,0,0,0,0,0,
↦ 0,1,0,0,0,0,0,
↦ 0,0,1,0,0,0,0,
↦ 0,0,0,1,0,0,0,
↦ 0,0,0,0,1,0,0,
↦ 0,0,0,0,0,1,0,
↦ 0,0,0,0,0,0,1
ext(3,I);   // too big to be displayed here
```

The library `homolog.lib` contains several procedures for computing Ext-modules and related modules, which are much more general and sophisticated than the above one. They are used in the following example:

If M is a module, then $\mathrm{Ext}^1(M,M)$, resp. $\mathrm{Ext}^2(M,M)$, are the modules of infinitesimal deformations, respectively of obstructions, of M (like T1 and T2 for a singularity). Similar to the treatment of singularities, the semiuniversal deformation of M can be computed (if Ext^1 is finite dimensional) with the help of Ext^1, Ext^2 and the cup product. There is an extra procedure for $\mathrm{Ext}^k(R/J,R)$ if J is an ideal in R , since this is faster than the general Ext.

We compute

- the infinitesimal deformations ($=\mathrm{Ext}^1(K,K)$) and obstructions ($=\mathrm{Ext}^2(K,K)$) of the residue field $K=R/m$ of an ordinary cusp, $R=K[x,y]_m/(x^2-y^3)$, $m=(x,y)$. To compute $\mathrm{Ext}^1(m,m)$ we have to apply `Ext(1,syz(m),syz(m))` with `syz(m)` the first syzygy module of m , which is isomorphic to $\mathrm{Ext}^2(K,K)$.
- $\mathrm{Ext}^k(R/i,R)$ for some ideal i and with an extra option.

```
  LIB "homolog.lib";
  ring R=0,(x,y),ds;
  ideal i=x2-y3;
  qring q = std(i);      // defines the quotient ring k[x,y]_m/(x2-y3)
  ideal m = maxideal(1);
  module T1K = Ext(1,m,m);  // computes Ext^1(R/m,R/m)
↦ // dimension of Ext^1:  0
↦ // vdim of Ext^1:       2
↦
  print(T1K);
↦ 0,x,0,y,
↦ x,0,y,0
  printlevel=2;             // gives more explanation
  module T2K=Ext(2,m,m);    // computes Ext^2(R/m,R/m)
```

```
↦ // Computing Ext^2 (help Ext; gives an explanation):
↦ // Let 0<--coker(M)<--F0<--F1<--F2<--... be a resolution of coker(M),
↦ // and 0<--coker(N)<--G0<--G1 a presentation of coker(N),
↦ // then Hom(F2,G0)-->Hom(F3,G0) is given by:
↦ y2,x,
↦ x, y
↦ // and Hom(F1,G0) + Hom(F2,G1)-->Hom(F2,G0) is given by:
↦ -y,x,  x,0,y,0,
↦ x, -y2,0,x,0,y
↦
↦ // dimension of Ext^2:  0
↦ // vdim of Ext^2:       2
↦
  print(std(T2K));
↦ 0,x,0,y,
↦ x,0,y,0
  printlevel=0;
  module E = Ext(1,syz(m),syz(m));
↦ // dimension of Ext^1:  0
↦ // vdim of Ext^1:       2
↦
  print(std(E));
↦ x, 0,-y2,x,0,y,
↦ -y,0,x,  0,y,0,
↦ 0, 1,0,  0,0,0,
↦ 1, 0,0,  0,0,0
  //The matrices which we have just computed are presentation matrices
  //of the modules T2K and E. Hence we may ignore those columns
  //containing 1 as an entry and see that T2K and E are isomorphic
  //as expected, but differently presented.
  //-------------------------------------------
  ring S=0,(x,y,z),dp;
  ideal  i = x2y,y2z,z3x;
  module E = Ext_R(2,i);
↦ // dimension of Ext^2:  1
↦
  print(E);
↦ 0,y,0,z2,
↦ z,0,0,-x,
↦ 0,0,x,-y
  // if a 3-rd argument of type int is given,
  // a list of Ext^k(R/i,R), a SB of Ext^k(R/i,R) and a vector space basis
  // is returned:
  list LE = Ext_R(3,i,0);
↦ // dimension of Ext^3:  0
↦ // vdim of Ext^3:       2
↦
  LE;
↦ [1]:
↦    _[1]=y*gen(1)
↦    _[2]=x*gen(1)
↦    _[3]=z2*gen(1)
↦ [2]:
```

```
↦    _[1]=y*gen(1)
↦    _[2]=x*gen(1)
↦    _[3]=z2*gen(1)
↦ [3]:
↦    _[1,1]=z
↦    _[1,2]=1
  print(LE[2]);
↦ y,x,z2
  print(kbase(LE[2]));
↦ z,1
```

A.3.7 Depth

We compute the depth of the module of Kaehler differentials $\mathrm{D}_k(\mathrm{R})$ of the variety defined by the $(m+1)$ -minors of a generic symmetric $(n \times n)$-matrix. We do this by computing the resolution over the polynomial ring. Then, by the Auslander-Buchsbaum formula, the depth is equal to the number of variables minus the length of a minimal resolution. This example was suggested by U. Vetter in order to check whether his bound $\mathrm{depth}(\mathrm{D}_k(R)) \geq m(m+1)/2+m-1$ could be improved.

```
  LIB "matrix.lib"; LIB "sing.lib";
  int n = 4;
  int m = 3;
  int N = n*(n+1) div 2;      // will become number of variables
  ring R = 32003,x(1..N),dp;
  matrix X = symmat(n);       // proc from matrix.lib
                              // creates the symmetric generic nxn matrix
  print(X);
↦ x(1),x(2),x(3),x(4),
↦ x(2),x(5),x(6),x(7),
↦ x(3),x(6),x(8),x(9),
↦ x(4),x(7),x(9),x(10)
  ideal J = minor(X,m);
  J=std(J);
  // Kaehler differentials D_k(R)
  // of R=k[x1..xn]/J:
  module D = J*freemodule(N)+transpose(jacob(J));
  ncols(D);
↦ 110
  nrows(D);
↦ 10
  //
  // Note: D is a submodule with 110 generators of a free module
  // of rank 10 over a polynomial ring in 10 variables.
  // Compute a full resolution of D with sres.
  // This takes about 17 sec on a Mac PB 520c and 2 sec an a HP 735
  int time = timer;
  module sD = std(D);
  list Dres = sres(sD,0);              // the full resolution
  timer-time;                          // time used for std + sres
↦ 0
  intmat B = betti(Dres);
  print(B,"betti");
↦            0     1     2     3     4     5     6
↦ ------------------------------------------------
```

```
↦       0:    10     -     -     -     -     -     -
↦       1:     -    10     -     -     -     -     -
↦       2:     -    84   144    60     -     -     -
↦       3:     -     -    35    80    60    16     1
↦ ------------------------------------------------
↦ total:    10    94   179   140    60    16     1
↦
  N-ncols(B)+1;                          // the desired depth
↦ 4
```

A.3.8 Factorization

The factorization of polynomials is implemented in the C++ libraries Factory (written mainly by Ruediger Stobbe) and libfac (written by Michael Messollen) which are part of the Singular system. For the factorization of univariate polynomials these libraries make use of the library NTL written by Victor Shoup.

```
  ring r = 0,(x,y),dp;
  poly f = 9x16-18x13y2-9x12y3+9x10y4-18x11y2+36x8y4
         +18x7y5-18x5y6+9x6y4-18x3y6-9x2y7+9y8;
  // = 9 * (x5-1y2)^2 * (x6-2x3y2-1x2y3+y4)
  factorize(f);
↦ [1]:
↦    _[1]=9
↦    _[2]=x6-2x3y2-x2y3+y4
↦    _[3]=-x5+y2
↦ [2]:
↦    1,1,2
  // returns factors and multiplicities,
  // first factor is a constant.
  poly g = (y4+x8)*(x2+y2);
  factorize(g);
↦ [1]:
↦    _[1]=1
↦    _[2]=x8+y4
↦    _[3]=x2+y2
↦ [2]:
↦    1,1,1
  // The same in characteristic 2:
  ring s = 2,(x,y),dp;
  poly g = (y4+x8)*(x2+y2);
  factorize(g);
↦ [1]:
↦    _[1]=1
↦    _[2]=x+y
↦    _[3]=x2+y
↦ [2]:
↦    1,2,4
  // factorization over algebraic extension fields
  ring rext = (0,i),(x,y),dp;
  minpoly = i2+1;
  poly g = (y4+x8)*(x2+y2);
  factorize(g);
↦ [1]:
```

```
↦     _[1]=1
↦     _[2]=x4+(-i)*y2
↦     _[3]=x4+(i)*y2
↦     _[4]=x+(i)*y
↦     _[5]=x+(-i)*y
↦  [2]:
↦     1,1,1,1,1
```

A.3.9 Primary decomposition

There are two algorithms implemented in Singular which provide primary decomposition: `primdecGTZ`, based on Gianni/Trager/Zacharias (written by Gerhard Pfister) and `primdecSY`, based on Shimoyama/Yokoyama (written by Wolfram Decker and Hans Schoenemann).

The result of `primdecGTZ` and `primdecSY` is returned as a list of pairs of ideals, where the second ideal is the prime ideal and the first ideal the corresponding primary ideal.

```
  LIB "primdec.lib";
  ring r = 0,(a,b,c,d,e,f),dp;
  ideal i= f3, ef2, e2f, bcf-adf, de+cf, be+af, e3;
  primdecGTZ(i);
↦ [1]:
↦    [1]:
↦       _[1]=f
↦       _[2]=e
↦    [2]:
↦       _[1]=f
↦       _[2]=e
↦ [2]:
↦    [1]:
↦       _[1]=f3
↦       _[2]=ef2
↦       _[3]=e2f
↦       _[4]=e3
↦       _[5]=de+cf
↦       _[6]=be+af
↦       _[7]=-bc+ad
↦    [2]:
↦       _[1]=f
↦       _[2]=e
↦       _[3]=-bc+ad
  // We consider now the ideal J of the base space of the
  // miniversal deformation of the cone over the rational
  // normal curve computed in section *8* and compute
  // its primary decomposition.
  ring R = 0,(A,B,C,D),dp;
  ideal J = CD, BD+D2, AD;
  primdecGTZ(J);
↦ [1]:
↦    [1]:
↦       _[1]=D
↦    [2]:
↦       _[1]=D
↦ [2]:
↦    [1]:
```

```
↦       _[1]=C
↦       _[2]=B+D
↦       _[3]=A
↦    [2]:
↦       _[1]=C
↦       _[2]=B+D
↦       _[3]=A
  // We see that there are two components which are both
  // prime, even linear subspaces, one 3-dimensional,
  // the other 1-dimensional.
  // (This is Pinkhams example and was the first known
  // surface singularity with two components of
  // different dimensions)
  //
  // Let us now produce an embedded component in the last
  // example, compute the minimal associated primes and
  // the radical. We use the Characteristic set methods
  // from primdec.lib.
  J = intersect(J,maxideal(3));
  // The following shows that the maximal ideal defines an embedded
  // (prime) component.
  primdecSY(J);
↦ [1]:
↦    [1]:
↦       _[1]=D
↦    [2]:
↦       _[1]=D
↦ [2]:
↦    [1]:
↦       _[1]=C
↦       _[2]=B+D
↦       _[3]=A
↦    [2]:
↦       _[1]=C
↦       _[2]=B+D
↦       _[3]=A
↦ [3]:
↦    [1]:
↦       _[1]=D2
↦       _[2]=C2
↦       _[3]=B2
↦       _[4]=AB
↦       _[5]=A2
↦       _[6]=BCD
↦       _[7]=ACD
↦    [2]:
↦       _[1]=D
↦       _[2]=C
↦       _[3]=B
↦       _[4]=A
  minAssChar(J);
↦ [1]:
↦    _[1]=C
```

```
↦    _[2]=B+D
↦    _[3]=A
↦ [2]:
↦    _[1]=D
  radical(J);
↦ _[1]=CD
↦ _[2]=BD+D2
↦ _[3]=AD
```

A.3.10 Normalization

The normalization will be computed for a reduced ring R/I . The result is a list of rings; ideals are always called `norid` in the rings of this list. The normalization of R/I is the product of the factor rings of the rings in the list divided out by the ideals `norid`.

```
  LIB "normal.lib";
  // ----- first example: rational quadruple point -----
  ring R=32003,(x,y,z),wp(3,5,15);
  ideal I=z*(y3-x5)+x10;
  list pr=normal(I);
↦
↦ // 'normal' created a list, say nor, of two elements.
↦ // To see the list type
↦       nor;
↦
↦ // * nor[1] is a list of 1 ring(s).
↦ // To access the i-th ring nor[1][i], give it a name, say Ri, and type
↦      def R1 = nor[1][1]; setring R1; norid; normap;
↦ // For the other rings type first (if R is the name of your base ring)
↦      setring R;
↦ // and then continue as for R1.
↦ // Ri/norid is the affine algebra of the normalization of R/P_i where
↦ // P_i is the i-th component of a decomposition of the input ideal id
↦ // and normap the normalization map from R to Ri/norid.
↦
↦ // * nor[2] is a list of 1 ideal(s). Let ci be the last generator
↦ // of the ideal nor[2][i]. Then the integral closure of R/P_i is
↦ // generated as R-submodule of the total ring of fractions by
↦ // 1/ci * nor[2][i].
  def S=pr[1][1];
  setring S;
  norid;
↦ norid[1]=T(2)*x+y*z
↦ norid[2]=T(1)*x^2-T(2)*y
↦ norid[3]=-T(1)*y+x^7-x^2*z
↦ norid[4]=T(1)*y^2*z+T(2)*x^8-T(2)*x^3*z
↦ norid[5]=T(1)^2+T(2)*z+x^4*y*z
↦ norid[6]=T(1)*T(2)+x^6*z-x*z^2
↦ norid[7]=T(2)^2+T(1)*x*z
↦ norid[8]=x^10-x^5*z+y^3*z
  // ----- second example: union of straight lines -----
  ring R1=0,(x,y,z),dp;
  ideal I=(x-y)*(x-z)*(y-z);
  list qr=normal(I);
```

```
↦
↦ // 'normal' created a list, say nor, of two elements.
↦ // To see the list type
↦       nor;
↦
↦ // * nor[1] is a list of 2 ring(s).
↦ // To access the i-th ring nor[1][i], give it a name, say Ri, and type
↦      def R1 = nor[1][1]; setring R1; norid; normap;
↦ // For the other rings type first (if R is the name of your base ring)
↦      setring R;
↦ // and then continue as for R1.
↦ // Ri/norid is the affine algebra of the normalization of R/P_i where
↦ // P_i is the i-th component of a decomposition of the input ideal id
↦ // and normap the normalization map from R to Ri/norid.
↦
↦ // * nor[2] is a list of 2 ideal(s). Let ci be the last generator
↦ // of the ideal nor[2][i]. Then the integral closure of R/P_i is
↦ // generated as R-submodule of the total ring of fractions by
↦ // 1/ci * nor[2][i].
  def S1=qr[1][1]; def S2=qr[1][2];
  setring S1; norid;
↦ norid[1]=-T(1)*y+T(1)*z+x-z
↦ norid[2]=T(1)*x-T(1)*y
↦ norid[3]=T(1)^2-T(1)
↦ norid[4]=x^2-x*y-x*z+y*z
  setring S2; norid;
↦ norid[1]=y-z
```

A.3.11 Kernel of module homomorphisms

Let A , B be two matrices of size $m \times r$ and $m \times s$ over the ring R and consider the corresponding maps

$$R^r \xrightarrow{A} R^m \xleftarrow{B} R^s \ .$$

We want to compute the kernel of the map $R^r \xrightarrow{A} R^m \longrightarrow R^m/\mathrm{Im}(B)$. This can be done using the `modulo` command:

$$\texttt{modulo}(A,B) = \ker(R^r \xrightarrow{A} R^m/\mathrm{Im}(B)) \ .$$

More precisely, the output of `modulo(A,B)` is a `module` such that the given generating `vector`s span the kernel on the right-hand side.

```
  ring r=0,(x,y,z),(c,dp);
  matrix A[2][2]=x,y,z,1;
  matrix B[2][2]=x2,y2,z2,xz;
  print(B);
↦ x2,y2,
↦ z2,xz
  def C=modulo(A,B);
  print(C);            // matrix of generators for the kernel
↦ yz2-x2, xyz-y2,  x2z-xy, x3-y2z,
↦ x2z-xz2,-x2z+y2z,xyz-yz2,0
  print(A*matrix(C));  // should be in Im(B)
↦ x2yz-x3,y3z-xy2, x3z+xy2z-y2z2-x2y,x4-xy2z,
↦ yz3-xz2,xyz2-x2z,x2z2-yz2,         x3z-y2z2
```

A.3.12 Algebraic dependence

Let g, $f_1, \ldots, f_r \in K[x_1, \ldots, x_n]$. We want to check whether

1. $f_1, \ldots, f_r$ are algebraically dependent.

   Let $I = \langle Y_1 - f_1, \ldots, Y_r - f_r \rangle \subseteq K[x_1, \ldots, x_n, Y_1, \ldots, Y_r]$. Then $I \cap K[Y_1, \ldots, Y_r]$ are the algebraic relations between $f_1, \ldots, f_r$.
2. $g \in K[f_1, \ldots, f_r]$.

   $g \in K[f_1, \ldots, f_r]$ if and only if the normal form of g with respect to I and a block ordering with respect to $X = (x_1, \ldots, x_n)$ and $Y = (Y_1, \ldots, Y_r)$ with $X > Y$ is in $K[Y]$.

Both questions can be answered using the following procedure. If the second argument is zero, it checks for algebraic dependence and returns the ideal of relations between the generators of the given ideal. Otherwise it checks for subring membership and returns the normal form of the second argument with respect to the ideal I.

```
proc algebraicDep(ideal J, poly g)
{
  def R=basering;         // give a name to the basering
  int n=size(J);
  int k=nvars(R);
  int i;
  intvec v;

  // construction of the new ring:

  // construct a weight vector
  v[n+k]=0;         // gives a zero vector of length n+k
  for(i=1;i<=k;i++)
  {
    v[i]=1;
  }
  string orde="(a("+string(v)+"),dp);";
  string ri="ring Rhelp=("+charstr(R)+"),
                        ("+varstr(R)+",Y(1.."+string(n)+")),"+orde;
                          // ring definition as a string
  execute(ri);            // execution of the string

  // construction of the new ideal I=(J[1]-Y(1),...,J[n]-Y(n))
  ideal I=imap(R,J);
  for(i=1;i<=n;i++)
  {
    I[i]=I[i]-var(k+i);
  }
  poly g=imap(R,g);
  if(g==0)
  {
    // construction of the ideal of relations by elimination
    poly el=var(1);
    for(i=2;i<=k;i++)
    {
      el=el*var(i);
    }
    ideal KK=eliminate(I,el);
```

```
    keepring(Rhelp);
    return(KK);
  }
  // reduction of g with respect to I
  ideal KK=reduce(g,std(I));
  keepring(Rhelp);
  return(KK);
}

// applications of the procedure
ring r=0,(x,y,z),dp;
ideal i=xz,yz;
algebraicDep(i,0);
↦ _[1]=0
// Note: after call of algebraicDep(), the basering is Rhelp.
setring r; kill Rhelp;
ideal j=xy+z2,z2+y2,x2y2-2xy3+y4;
algebraicDep(j,0);
↦ _[1]=Y(1)^2-2*Y(1)*Y(2)+Y(2)^2-Y(3)
setring r; kill Rhelp;
poly g=y2z2-xz;
algebraicDep(i,g);
↦ _[1]=Y(2)^2-Y(1)
// this shows that g is contained in i.
setring r; kill Rhelp;
algebraicDep(j,g);
↦ _[1]=-z^4+z^2*Y(2)-x*z
// this shows that g is contained in j.
```

A.4 Singularity Theory

A.4.1 Milnor and Tjurina number

The Milnor number, resp. the Tjurina number, of a power series f in $K[[x_1,\ldots,x_n]]$ is

$$\mathrm{milnor}(f) = \dim_K(K[[x_1,\ldots,x_n]]/\mathrm{jacob}(f)),$$

respectively

$$\mathrm{tjurina}(f) = \dim_K(K[[x_1,\ldots,x_n]]/((f)+\mathrm{jacob}(f)))$$

where `jacob(f)` is the ideal generated by the partials of `f`. `tjurina(f)` is finite, if and only if `f` has an isolated singularity. The same holds for `milnor(f)` if K has characteristic 0. Singular displays -1 if the dimension is infinite.

Singular cannot compute with infinite power series. But it can work in $\mathrm{Loc}_{(x)}K[x_1,\ldots,x_n]$, the localization of $K[x_1,\ldots,x_n]$ at the maximal ideal $(x_1,\ldots,x_n)$. To do this, one has to define a ring with a local monomial ordering such as ds, Ds, ls, ws, Ws (the second letter 's' referring to power 's'eries), or an appropriate matrix ordering. See Section B.2 [Monomial orderings], page 696 for a menu of possible orderings.

For theoretical reasons, the vector space dimension computed over the localization ring coincides with the Milnor (resp. Tjurina) number as defined above (in the power series ring).

We show in the example below the following:

- set option `prot` to have a short protocol during standard basis computation
- define the ring `r1` of characteristic 32003 with variables `x,y,z`, monomial ordering `ds`, series ring (i.e., K[x,y,z] localized at (x,y,z))
- list the information about `r1` by typing its name
- define the integers `a,b,c,t`
- define a polynomial `f` (depending on `a,b,c,t`) and display it
- define the jacobian ideal `i` of `f`
- compute a standard basis of `i`
- compute the Milnor number (=250) with `vdim` and create and display a string in order to comment the result (text between quotes `" "`; is a 'string')
- compute a standard basis of `i+(f)`
- compute the Tjurina number (=195) with `vdim`
- then compute the Milnor number (=248) and the Tjurina number (=195) for `t`=1
- reset the option to `noprot`

See also Section D.6.15 [sing_lib], page 783 for the library commands for the computation of the Milnor and Tjurina number.

```
  option(prot);
  ring r1 = 32003,(x,y,z),ds;
  r1;
↦ // coefficients: ZZ/32003
↦ // number of vars : 3
↦ //        block   1 : ordering ds
↦ //                  : names    x y z
↦ //        block   2 : ordering C
  int a,b,c,t=11,5,3,0;
  poly f = x^a+y^b+z^(3*c)+x^(c+2)*y^(c-1)+x^(c-1)*y^(c-1)*z3+
           x^(c-2)*y^c*(y^2+t*x)^2;
  f;
↦ y5+x5y2+x2y2z3+xy7+z9+x11
  ideal i=jacob(f);
  i;
↦ i[1]=5x4y2+2xy2z3+y7+11x10
↦ i[2]=5y4+2x5y+2x2yz3+7xy6
↦ i[3]=3x2y2z2+9z8
  ideal j=std(i);
↦ 7(2)s8s10s11s12s(3)s13(4)s(5)s14(6)s(7)15--.s(6)-16.-.s(5)17.s(7)s--s18(6\
   ).--19-..sH(24)20(3)...21....22....23.--24-
↦ product criterion:10 chain criterion:69
  "The Milnor number of f(11,5,3) for t=0 is", vdim(j);
↦ The Milnor number of f(11,5,3) for t=0 is 250
  j=i+f;    // override j
  j=std(j);
↦ 7(3)s8(2)s10s11(3)ss12(4)s(5)s13(6)s(8)s14(9).s(10).15--sH(23)(8)...16...\
   ...17.......sH(21)(9)sH(20)16(10).17...........18.......19..----..sH(19)
↦ product criterion:10 chain criterion:53
  vdim(j);  // compute the Tjurina number for t=0
↦ 195
  t=1;
  f=x^a+y^b+z^(3*c)+x^(c+2)*y^(c-1)+x^(c-1)*y^(c-1)*z3
```

```
  +x^(c-2)*y^c*(y^2+t*x)^2;
ideal i1=jacob(f);
ideal j1=std(i1);
↦ 7(2)s8s10s11s12s13(3)ss(4)s14(5)s(6)s15(7).....s(8)16.s...s(9)..17.......\
  .....s18(10).....s(11)..-.19.......sH(24)(10).....20...........21........\
  ..22............................23..............................24.----\
  ------.25.26
↦ product criterion:11 chain criterion:83
"The Milnor number of f(11,5,3) for t=1:",vdim(j1);
↦ The Milnor number of f(11,5,3) for t=1: 248
vdim(std(j1+f));   // compute the Tjurina number for t=1
↦ 7(16)s8(15)s10s11ss(16)-12.s-s13s(17)s(18)s(19)-s(18).-14-s(17)-s(16)ss(1\
  7)s15(18)..-s...--.16....-.......s(16).sH(23)s(18)...17..........18......\
  ...sH(20)17(17)...................18..........19..---....-.-.........20.\
  -----...s17(9).........18..............19..-.......20.-......21.........s\
  H(19)16(5).....18......19.-----
↦ product criterion:15 chain criterion:174
↦ 195
option(noprot);
```

A.4.2 Critical points

The same computation which computes the Milnor, resp. the Tjurina, number, but with ordering `dp` instead of `ds` (i.e., in $K[x_1,\ldots,x_n]$ instead of $\mathrm{Loc}_{(x)}K[x_1,\ldots,x_n]$) gives:

- the number of critical points of `f` in the affine space (counted with multiplicities)
- the number of singular points of `f` on the affine hypersurface `f`=0 (counted with multiplicities).

We start with the ring `r1` from section Section A.4.1 [Milnor and Tjurina number], page 662 and its elements.

The following will be implemented below:

- reset the protocol option and activate the timer
- define the ring `r2` of characteristic 32003 with variables `x,y,z` and monomial ordering `dp` (= degrevlex) (i.e., the polynomial ring = K[x,y,z]).
- Note that polynomials, ideals, matrices (of polys), vectors, modules belong to a ring, hence we have to define `f` and `jacob(f)` again in `r2`. Since these objects are local to a ring, we may use the same names. Instead of defining `f` again we map it from ring `r1` to `r2` by using the `imap` command (`imap` is a convenient way to map variables from some ring identically to variables with the same name in the basering, even if the ground field is different. Compare with `fetch` which works for almost identical rings, e.g., if the rings differ only by the ordering or by the names of the variables and which may be used to rename variables). Integers and strings, however, do not belong to any ring. Once defined they are globally known.
- The result of the computation here (together with the previous one in Section A.4.1 [Milnor and Tjurina number], page 662) shows that (for `t`=0) $\dim_K(\mathrm{Loc}_{(x,y,z)}K[x,y,z]/\mathrm{jacob}(f))$ = 250 (previously computed) while $\dim_K(K[x,y,z]/\mathrm{jacob}(f)) = 536$. Hence `f` has 286 critical points, counted with multiplicity, outside the origin. Moreover, since $\dim_K(\mathrm{Loc}_{(x,y,z)}K[x,y,z]/(\mathrm{jacob}(f)+(f))) = 195 = \dim_K(K[x,y,z]/(\mathrm{jacob}(f)+(f)))$, the affine surface `f`=0 is smooth outside the origin.

```
ring r1 = 32003,(x,y,z),ds;
int a,b,c,t=11,5,3,0;
poly f = x^a+y^b+z^(3*c)+x^(c+2)*y^(c-1)+x^(c-1)*y^(c-1)*z3+
         x^(c-2)*y^c*(y^2+t*x)^2;
```

```
  option(noprot);
  timer=1;
  ring r2 = 32003,(x,y,z),dp;
  poly f=imap(r1,f);
  ideal j=jacob(f);
  vdim(std(j));
↦ 536
  vdim(std(j+f));
↦ 195
  timer=0;  // reset timer
```

A.4.3 Polar curves

The polar curve of a hypersurface given by a polynomial $f \in k[x_1,\dots,x_n,t]$ with respect to t (we may consider $f = 0$ as a family of hypersurfaces parametrized by t) is defined as the Zariski closure of $V(\partial f/\partial x_1,\dots,\partial f/\partial x_n) \setminus V(f)$ if this happens to be a curve. Some authors consider $V(\partial f/\partial x_1,\dots,\partial f/\partial x_n)$ itself as polar curve.

We may consider projective hypersurfaces (in P^n), affine hypersurfaces (in k^n) or germs of hypersurfaces (in $(k^n,0)$), getting in this way projective, affine or local polar curves.

Now let us compute this for a family of curves. We need the library `elim.lib` for saturation and `sing.lib` for the singular locus.

```
  LIB "elim.lib";
  LIB "sing.lib";
  // Affine polar curve:
  ring R = 0,(x,z,t),dp;               // global ordering dp
  poly f = z5+xz3+x2-tz6;
  dim_slocus(f);                       // dimension of singular locus
↦ 1
  ideal j = diff(f,x),diff(f,z);
  dim(std(j));                         // dim V(j)
↦ 1
  dim(std(j+ideal(f)));                // V(j,f) also 1-dimensional
↦ 1
  // j defines a curve, but to get the polar curve we must remove the
  // branches contained in f=0 (they exist since dim V(j,f) = 1). This
  // gives the polar curve set theoretically. But for the structure we
  // may take either j:f or j:f^k for k sufficiently large. The first is
  // just the ideal quotient, the second the iterated ideal quotient
  // or saturation. In our case both coincide.
  ideal q = quotient(j,ideal(f));     // ideal quotient
  ideal qsat = sat(j,f)[1];           // saturation, proc from elim.lib
  ideal sq = std(q);
  dim(sq);
↦ 1
  // 1-dimensional, hence q defines the affine polar curve
  //
  // to check that q and qsat are the same, we show both inclusions, i.e.,
  // both reductions must give the 0-ideal
  size(reduce(qsat,sq));
↦ 0
  size(reduce(q,std(qsat)));
↦ 0
```

```
  qsat;
↦ qsat[1]=12zt+3z-10
↦ qsat[2]=5z2+12xt+3x
↦ qsat[3]=144xt2+72xt+9x+50z
  // We see that the affine polar curve does not pass through the origin,
  // hence we expect the local polar "curve" to be empty
  // ------------------------------------------------
  // Local polar curve:
  ring r = 0,(x,z,t),ds;              // local ordering ds
  poly f = z5+xz3+x2-tz6;
  ideal j = diff(f,x),diff(f,z);
  dim(std(j));                        // V(j) 1-dimensional
↦ 1
  dim(std(j+ideal(f)));               // V(j,f) also 1-dimensional
↦ 1
  ideal q = quotient(j,ideal(f));     // ideal quotient
  q;
↦ q[1]=1
  // The local polar "curve" is empty, i.e., V(j) is contained in V(f)
  // ------------------------------------------------
  // Projective polar curve: (we need "sing.lib" and "elim.lib")
  ring P = 0,(x,z,t,y),dp;            // global ordering dp
  poly f = z5y+xz3y2+x2y4-tz6;
                                      // but consider t as parameter
  dim_slocus(f);              // projective 1-dimensional singular locus
↦ 2
  ideal j = diff(f,x),diff(f,z);
  dim(std(j));                        // V(j), projective 1-dimensional
↦ 2
  dim(std(j+ideal(f)));               // V(j,f) also projective 1-dimensional
↦ 2
  ideal q = quotient(j,ideal(f));
  ideal qsat = sat(j,f)[1];           // saturation, proc from elim.lib
  dim(std(qsat));
↦ 2
  // projective 1-dimensional, hence q and/or qsat define the projective
  // polar curve. In this case, q and qsat are not the same, we needed
  // 2 quotients.
  // Let us check both reductions:
  size(reduce(qsat,std(q)));
↦ 4
  size(reduce(q,std(qsat)));
↦ 0
  // Hence q is contained in qsat but not conversely
  q;
↦ q[1]=12zty+3zy-10y2
↦ q[2]=60z2t-36xty-9xy-50zy
↦ q[3]=12xty2+5z2y+3xy2
↦ q[4]=z3y+2xy3
  qsat;
↦ qsat[1]=12zt+3z-10y
↦ qsat[2]=12xty+5z2+3xy
↦ qsat[3]=144xt2+72xt+9x+50z
```

```
↦ qsat[4]=z3+2xy2
  //
  // Now consider again the affine polar curve,
  // homogenize it with respect to y (deg t=0) and compare:
  // affine polar curve:
  ideal qa = 12zt+3z-10,5z2+12xt+3x,-144xt2-72xt-9x-50z;
  // homogenized:
  ideal qh = 12zt+3z-10y,5z2+12xyt+3xy,-144xt2-72xt-9x-50z;
  size(reduce(qh,std(qsat)));
↦ 0
  size(reduce(qsat,std(qh)));
↦ 0
  // both ideals coincide
```

A.4.4 T1 and T2

T^1 , resp. T^2 , of an ideal j usually denote the modules of infinitesimal deformations, resp. of obstructions. In SINGULAR there are procedures `T_1` and `T_2` in `sing.lib` such that `T_1(j)` and `T_2(j)` compute a standard basis of a presentation of these modules. If T^1, T_2 are finite dimensional K-vector spaces (e.g., for isolated singularities), a basis can be computed by applying `kbase(T_1(j));`, resp. `kbase(T_2(j));`, the dimensions by applying `vdim`. For a complete intersection j the procedure `Tjurina` also computes T^1 , but faster ($T^2 = 0$ in this case). For a non complete intersection, it is faster to use the procedure `T_12` instead of `T_1` and `T_2`. Type `help T_1;` (or `help T_2;` or `help T_12;`) to obtain more detailed information about these procedures.

We give three examples, the first being a hypersurface, the second a complete intersection, the third not a complete intersection:

- load `sing.lib`
- check whether the ideal j is a complete intersection. It is, if number of variables = dimension + minimal number of generators
- compute the Tjurina number
- compute a vector space basis (kbase) of T^1
- compute the Hilbert function of T^1
- create a polynomial encoding the Hilbert series
- compute the dimension of T^2

```
  LIB "sing.lib";
  ring R=32003,(x,y,z),ds;
  // ---------------------------------------
  // hypersurface case (from series T[p,q,r]):
  int p,q,r = 3,3,4;
  poly f = x^p+y^q+z^r+xyz;
  tjurina(f);
↦ 8
  // Tjurina number = 8
  kbase(Tjurina(f));
↦ // Tjurina number = 8
↦ _[1]=z3
↦ _[2]=z2
↦ _[3]=yz
↦ _[4]=xz
↦ _[5]=z
```

```
↦ _[6]=y
↦ _[7]=x
↦ _[8]=1
  // ---------------------------------------
  // complete intersection case (from series P[k,l]):
  int k,l =3,2;
  ideal j=xy,x^k+y^l+z2;
  dim(std(j));          // Krull dimension
↦ 1
  size(minbase(j));     // minimal number of generators
↦ 2
  tjurina(j);           // Tjurina number
↦ 6
  module T=Tjurina(j);
↦ // Tjurina number = 6
  kbase(T);             // a sparse output of the k-basis of T_1
↦ _[1]=z*gen(1)
↦ _[2]=gen(1)
↦ _[3]=y*gen(2)
↦ _[4]=x2*gen(2)
↦ _[5]=x*gen(2)
↦ _[6]=gen(2)
  print(kbase(T));      // columns of matrix are a k-basis of T_1
↦ z,1,0,0, 0,0,
↦ 0,0,y,x2,x,1
  // ---------------------------------------
  // general case (cone over rational normal curve of degree 4):
  ring r1=0,(x,y,z,u,v),ds;
  matrix m[2][4]=x,y,z,u,y,z,u,v;
  ideal i=minor(m,2);   // 2x2 minors of matrix m
  module M=T_1(i);       // a presentation matrix of T_1
↦ // dim T_1 = 4
  vdim(M);              // Tjurina number
↦ 4
  hilb(M);              // display of both Hilbert series
↦ //         4 t^0
↦ //       -20 t^1
↦ //        40 t^2
↦ //       -40 t^3
↦ //        20 t^4
↦ //        -4 t^5
↦
↦ //         4 t^0
↦ // dimension (local)   = 0
↦ // multiplicity = 4
  intvec v1=hilb(M,1);  // first Hilbert series as intvec
  intvec v2=hilb(M,2);  // second Hilbert series as intvec
  v1;
↦ 4,-20,40,-40,20,-4,0
  v2;
↦ 4,0
  v1[3];                // 3rd coefficient of the 1st Hilbert series
↦ 40
```

```
  module N=T_2(i);
↦ // dim T_2 = 3

// In some cases it might be useful to have a polynomial in some ring
// encoding the Hilbert series. This polynomial can then be
// differentiated, evaluated etc. It can be done as follows:
ring H = 0,t,ls;
poly h1;
int ii;
for (ii=1; ii<=size(v1); ii=ii+1)
{
   h1=h1+v1[ii]*t^(ii-1);
}
h1;                   // 1st Hilbert series
↦ 4-20t+40t2-40t3+20t4-4t5
diff(h1,t);           // differentiate  h1
↦ -20+80t-120t2+80t3-20t4
subst(h1,t,1);        // substitute t by 1
↦ 0

// The procedures T_1, T_2, T_12 may be called with two arguments and then
// they return a list with more information (type help T_1; etc.)
// e.g., T_12(i,<any>); returns a list with 9 nonempty objects where
// _[1] = std basis of T_1-module, _[2] = std basis of T_2-module,
// _[3]= vdim of T_1, _[4]= vdim of T_2
setring r1;           // make r1 again the basering
list L = T_12(i,1);
↦ // dim T_1  =  4
↦ // dim T_2  =  3
kbase(L[1]);          // kbase of T_1
↦ _[1]=1*gen(2)
↦ _[2]=1*gen(3)
↦ _[3]=1*gen(6)
↦ _[4]=1*gen(7)
kbase(L[2]);          // kbase of T_2
↦ _[1]=1*gen(6)
↦ _[2]=1*gen(8)
↦ _[3]=1*gen(9)
L[3];                 // vdim of T_1
↦ 4
L[4];                 // vdim of T_2
↦ 3
```

A.4.5 Deformations

- The libraries `sing.lib`, respectively `deform.lib`, contain procedures to compute total and base space of the miniversal (= semiuniversal) deformation of an isolated complete intersection singularity, respectively of an arbitrary isolated singularity.
- The procedure `deform` in `sing.lib` returns a matrix whose columns $h_1, \ldots, h_r$ represent all 1st order deformations. More precisely, if $I \subset R$ is the ideal generated by $f_1, ..., f_s$, then any infinitesimal deformation of R/I over $K[\varepsilon]/(\varepsilon^2)$ is given by $f + \varepsilon g$, where $f = (f_1, ..., f_s)$, and where g is a K-linear combination of the h_i.

- The procedure `versal` in `deform.lib` computes a formal miniversal deformation up to a certain order which can be prescribed by the user. For a complete intersection the 1st order part is already miniversal.
- The procedure `versal` extends the basering to a new ring with additional deformation parameters which contains the equations for the miniversal base space and the miniversal total space.
- There are default names for the objects created, but the user may also choose their own names.
- If the user sets `printlevel=2;` before running `versal`, some intermediate results are shown. This is useful since `versal` is already complicated and might run for some time on more complicated examples. (type `help versal;`)

We compute for the same examples as in the section Section A.4.4 [T1 and T2], page 667 the miniversal deformations:

```
  LIB "deform.lib";
  ring R=32003,(x,y,z),ds;
  //----------------------------------------------------
  // hypersurface case (from series T[p,q,r]):
  int p,q,r = 3,3,4;
  poly f = x^p+y^q+z^r+xyz;
  print(deform(f));
↦ z3,z2,yz,xz,z,y,x,1
  // the miniversal deformation of f=0 is the projection from the
  // miniversal total space to the miniversal base space:
  // { (A,B,C,D,E,F,G,H,x,y,z) | x3+y3+xyz+z4+A+Bx+Cxz+Dy+Eyz+Fz+Gz2+Hz3 =0 }
  //  --> { (A,B,C,D,E,F,G,H) }
  //----------------------------------------------------
  // complete intersection case (from series P[k,l]):
  int k,l =3,2;
  ideal j=xy,x^k+y^l+z2;
  print(deform(j));
↦ 0,0, 0,0,z,1,
↦ y,x2,x,1,0,0
  def L=versal(j);            // using default names
↦ // smooth base space
↦ // ready: T_1 and T_2
↦ 
↦ 
↦ // 'versal' returned a list, say L, of four rings. In L[1] are stored:
↦ //   as matrix Fs: Equations of total space of the miniversal deformation\
   ,
↦ //   as matrix Js: Equations of miniversal base space,
↦ //   as matrix Rs: syzygies of Fs mod Js.
↦ // To access these data, type
↦      def Px=L[1]; setring Px; print(Fs); print(Js); print(Rs);
↦ 
↦ // L[2] = L[1]/Fo extending Qo=Po/Fo,
↦ // L[3] = the embedding ring of the versal base space,
↦ // L[4] = L[1]/Js extending L[3]/Js.
↦ 
  def Px=L[1]; setring Px;
  show(Px);                   // show is a procedure from inout.lib
↦ // ring: (ZZ/32003),(A,B,C,D,E,F,x,y,z),(ds(6),ds(3),C,L(1048575));
```

```
↦ // minpoly = 0
↦ // objects belonging to this ring:
↦ // Rs                            [0]  matrix 2 x 1
↦ // Fs                            [0]  matrix 1 x 2
↦ // Js                            [0]  matrix 1 x 0
  listvar(matrix);
↦ // Rs                            [0]  matrix 2 x 1
↦ // Fs                            [0]  matrix 1 x 2
↦ // Js                            [0]  matrix 1 x 0
  // ___ Equations of miniversal base space ___:
  Js;
↦
  // ___ Equations of miniversal total space ___:
  Fs;
↦ Fs[1,1]=y2+z2+x3+Cy+Dx2+Ex+F
↦ Fs[1,2]=xy+Az+B
  // the miniversal deformation of V(j) is the projection from the
  // miniversal total space to the miniversal base space:
  // { (A,B,C,D,E,F,x,y,z) | xy+F+Ez=0, y2+z2+x3+D+Cx+Bx2+Ay=0 }
  //  --> { (A,B,C,D,E,F) }
  //----------------------------------------------------
  // general case (cone over rational normal curve of degree 4):
  kill L;
  ring r1=0,(x,y,z,u,v),ds;
  matrix m[2][4]=x,y,z,u,y,z,u,v;
  ideal i=minor(m,2);                // 2x2 minors of matrix m
  int time=timer;
  // Call parameters of the miniversal base A(1),A(2),...:
  def L=versal(i,0,"","A(");
↦ // ready: T_1 and T_2
↦ // start computation in degree 2.
↦
↦
↦ // 'versal' returned a list, say L, of four rings. In L[1] are stored:
↦ //   as matrix Fs: Equations of total space of the miniversal deformation\
   ,
↦ //   as matrix Js: Equations of miniversal base space,
↦ //   as matrix Rs: syzygies of Fs mod Js.
↦ // To access these data, type
↦      def Px=L[1]; setring Px; print(Fs); print(Js); print(Rs);
↦
↦ // L[2] = L[1]/Fo extending Qo=Po/Fo,
↦ // L[3] = the embedding ring of the versal base space,
↦ // L[4] = L[1]/Js extending L[3]/Js.
↦
  "// used time:",timer-time,"sec";   // time of last command
↦ // used time: 0 sec
  def Def_rPx=L[1]; setring Def_rPx;
  Fs;
↦ Fs[1,1]=u^2-z*v-A(2)*u+A(4)*v
↦ Fs[1,2]=z*u-y*v-A(1)*u+A(4)*u
↦ Fs[1,3]=y*u-x*v+A(3)*u+A(4)*z
↦ Fs[1,4]=z^2-y*u-A(1)*z+A(2)*y
```

```
↦ Fs[1,5]=y*z-x*u+A(2)*x+A(3)*z
↦ Fs[1,6]=y^2-x*z+A(1)*x+A(3)*y
  Js;
↦ Js[1,1]=A(2)*A(4)
↦ Js[1,2]=-A(1)*A(4)+A(4)^2
↦ Js[1,3]=A(3)*A(4)
  // the miniversal deformation of V(i) is the projection from the
  // miniversal total space to the miniversal base space:
  // { (A(1..4),x,y,z,u,v) |
  //         -u^2+x*v+A(2)*u+A(4)*v=0, -z*u+y*v-A(1)*u+A(3)*u=0,
  //         -y*u+x*v+A(3)*u+A(4)*z=0,  z^2-y*u+A(1)*z+A(2)*y=0,
  //          y*z-x*u+A(2)*x-A(3)*z=0, -y^2+x*z+A(1)*x+A(3)*y=0 }
  //  --> { A(1..4) |
  //          A(2)*A(4) = -A(3)*A(4) = -A(1)*A(4)+A(4)^2 = 0 }
  //----------------------------------------------------
```

A.4.6 Invariants of plane curve singularities

The Puiseux pairs of an irreducible and reduced plane curve singularity are probably its most important invariants. They can be computed from its Hamburger-Noether expansion (which is the analogue of the Puiseux expansion in characteristic 0 for fields of arbitrary characteristic).

The library `hnoether.lib` (see Section D.6.11 [hnoether_lib], page 780) uses the algorithm of Antonio Campillo in "Algebroid curves in positive characteristic" SLN 813, 1980. This algorithm has the advantage that it needs least possible field extensions and, moreover, works in any characteristic. This fact can be used to compute the invariants over a field of finite characteristic, say 32003, which will most probably be the same as in characteristic 0.

We compute the Hamburger-Noether expansion of a plane curve singularity given by a polynomial `f` in two variables. This expansion is given by a matrix, and it allows us to compute a primitive parametrization (up to a given order) for the curve singularity defined by `f` and numerical invariants such as the

- characteristic exponents,
- Puiseux pairs (of a complex model),
- degree of the conductor,
- delta invariant,
- generators of the semigroup.

Besides commands for computing a parametrization and the invariants mentioned above, the library `hnoether.lib` provides commands for the computation of the Newton polygon of `f`, the square-free part of `f` and a procedure to convert one set of invariants to another.

```
  LIB "hnoether.lib";
  // ======== The irreducible case ========
  ring s = 0,(x,y),ds;
  poly f = y4-2x3y2-4x5y+x6-x7;
  list hn = develop(f);
  show(hn[1]);     // Hamburger-Noether matrix
↦ // matrix, 3x3
↦ 0,x,  0,
↦ 0,1,  x,
↦ 0,1/4,-1/2
  displayHNE(hn);  // Hamburger-Noether development
↦   y = z(1)*x
```

```
↦   x = z(1)^2+z(1)^2*z(2)
↦   z(1) = 1/4*z(2)^2-1/2*z(2)^3 + ..... (terms of degree >=4)
  setring s;
  displayInvariants(hn);
↦  characteristic exponents  : 4,6,7
↦  generators of semigroup   : 4,6,13
↦  Puiseux pairs             : (3,2)(7,2)
↦  degree of the conductor   : 16
↦  delta invariant           : 8
↦  sequence of multiplicities: 4,2,2,1,1
  // invariants(hn);  returns the invariants as list
  // partial parametrization of f: param takes the first variable
  // as infinite except the ring has more than 2 variables. Then
  // the 3rd variable is chosen.
  param(hn);
↦ // ** Warning: result is exact up to order 5 in x and 7 in y !
↦ _[1]=1/16x4-3/16x5+1/4x7
↦ _[2]=1/64x6-5/64x7+3/32x8+1/16x9-1/8x10
  ring extring=0,(x,y,t),ds;
  poly f=x3+2xy2+y2;
  list hn=develop(f,-1);
  param(hn);       // partial parametrization of f
↦ // ** Warning: result is exact up to order 2 in x and 3 in y !
↦ _[1]=-t2
↦ _[2]=-t3
  list hn1=develop(f,6);
  param(hn1);     // a better parametrization
↦ // ** Warning: result is exact up to order 6 in x and 7 in y !
↦ _[1]=-t2+2t4-4t6
↦ _[2]=-t3+2t5-4t7
  // instead of recomputing you may extend the development:
  list hn2=extdevelop(hn,12);
  param(hn2);     // a still better parametrization
↦ // ** Warning: result is exact up to order 12 in x and 13 in y !
↦ _[1]=-t2+2t4-4t6+8t8-16t10+32t12
↦ _[2]=-t3+2t5-4t7+8t9-16t11+32t13
  //
  // ======== The reducible case ========
  ring r = 0,(x,y),dp;
  poly f=x11-2y2x8-y3x7-y2x6+y4x5+2y4x3+y5x2-y6;
  // = (x5-1y2) * (x6-2x3y2-1x2y3+y4)
  list L=hnexpansion(f);
↦ // No change of ring necessary, return value is HN expansion.
  show(L[1][1]);     // Hamburger-Noether matrix of 1st branch
↦ // matrix, 3x3
↦ 0,x,0,
↦ 0,1,x,
↦ 0,1,-1
  displayInvariants(L);
↦  --- invariants of branch number 1 : ---
↦  characteristic exponents  : 4,6,7
↦  generators of semigroup   : 4,6,13
↦  Puiseux pairs             : (3,2)(7,2)
```

```
↦  degree of the conductor   : 16
↦  delta invariant           : 8
↦  sequence of multiplicities: 4,2,2,1,1
↦
↦  --- invariants of branch number 2 : ---
↦  characteristic exponents  : 2,5
↦  generators of semigroup   : 2,5
↦  Puiseux pairs             : (5,2)
↦  degree of the conductor   : 4
↦  delta invariant           : 2
↦  sequence of multiplicities: 2,2,1,1
↦
↦  -------------- contact numbers : --------------
↦
↦ branch |    2
↦ -------+-----
↦     1  |    2
↦
↦  -------------- intersection multiplicities : --------------
↦
↦ branch |    2
↦ -------+-----
↦     1  |   12
↦
↦  -------------- delta invariant of the curve :  22
  param(L[2]);       // parametrization of 2nd branch
↦ _[1]=x2
↦ _[2]=x5
```

A.4.7 Branches of space curve singularities

In this example, the number of branches of a given quasihomogeneous isolated space curve singularity will be computed as an example of the pitfalls appearing in the use of primary decomposition. When dealing with singularities, two situations are possible in which the primary decomposition algorithm might not lead to a complete decomposition: first of all, one of the computed components could be globally irreducible, but analytically reducible (this is impossible for quasihomogeneous singularities) and, as a second possibility, a component might be irreducible over the rational numbers, but reducible over the complex numbers.

```
  ring r=0,(x,y,z),ds;
  ideal i=x^4-y*z^2,x*y-z^3,y^2-x^3*z;  // the space curve singularity
  qhweight(i);
↦ 1,2,1
  // The given space curve singularity is quasihomogeneous. Hence we can pass
  // to the polynomial ring.
  ring rr=0,(x,y,z),dp;
  ideal i=imap(r,i);
  resolution ires=mres(i,0);
  ires;
↦   1       3       2
↦ rr <--  rr <--  rr
↦
↦ 0       1       2
↦
```

```
// From the structure of the resolution, we see that the Cohen-Macaulay
// type of the given singularity is 2
//
// Let us now look for the branches using the primdec library.
LIB "primdec.lib";
primdecSY(i);
↦ [1]:
↦    [1]:
↦       _[1]=z3-xy
↦       _[2]=x3+x2z+xz2+xy+yz
↦       _[3]=x2z2+x2y+xyz+yz2+y2
↦    [2]:
↦       _[1]=z3-xy
↦       _[2]=x3+x2z+xz2+xy+yz
↦       _[3]=x2z2+x2y+xyz+yz2+y2
↦ [2]:
↦    [1]:
↦       _[1]=x-z
↦       _[2]=z2-y
↦    [2]:
↦       _[1]=x-z
↦       _[2]=z2-y
def li=_[1];
ideal i2=li[2];       // call the first ideal i1
// The curve seems to have 2 branches by what we computed using the
// algorithm of Shimoyama-Yokoyama.
// Now the same computation by the Gianni-Trager-Zacharias algorithm:
primdecGTZ(i);
↦ [1]:
↦    [1]:
↦       _[1]=-z2+y
↦       _[2]=x-z
↦    [2]:
↦       _[1]=-z2+y
↦       _[2]=x-z
↦ [2]:
↦    [1]:
↦       _[1]=z8+yz6+y2z4+y3z2+y4
↦       _[2]=xz5+z6+yz4+y2z2+y3
↦       _[3]=-z3+xy
↦       _[4]=x2z2+xz3+xyz+yz2+y2
↦       _[5]=x3+x2z+xz2+xy+yz
↦    [2]:
↦       _[1]=z8+yz6+y2z4+y3z2+y4
↦       _[2]=xz5+z6+yz4+y2z2+y3
↦       _[3]=-z3+xy
↦       _[4]=x2z2+xz3+xyz+yz2+y2
↦       _[5]=x3+x2z+xz2+xy+yz
// Having computed the primary decomposition in 2 different ways and
// having obtained the same number of branches, we might expect that the
// number of branches is really 2, but we can check this by formulae
// for the invariants of space curve singularities:
//
```

```
// mu = tau - t + 1 (for quasihomogeneous curve singularities)
// where mu denotes the Milnor number, tau the Tjurina number and
// t the Cohen-Macaulay type
//
// mu = 2 delta - r + 1
// where delta denotes the delta-Invariant and r the number of branches
//
// tau can be computed by using the corresponding procedure T1 from
// sing.lib.
setring r;
LIB "sing.lib";
T_1(i);
↦ // dim T_1 = 13
↦ _[1]=gen(6)+2z*gen(5)
↦ _[2]=gen(4)+3x2*gen(2)
↦ _[3]=gen(3)+gen(1)
↦ _[4]=x*gen(5)-y*gen(2)-z*gen(1)
↦ _[5]=x*gen(1)-z2*gen(2)
↦ _[6]=y*gen(5)+3x2z*gen(2)
↦ _[7]=y*gen(2)-z*gen(1)
↦ _[8]=2y*gen(1)-z2*gen(5)
↦ _[9]=z2*gen(5)
↦ _[10]=z2*gen(1)
↦ _[11]=x3*gen(2)
↦ _[12]=x2z2*gen(2)
↦ _[13]=xz3*gen(2)
↦ _[14]=z4*gen(2)
setring rr;
// Hence tau is 13 and therefore mu is 12. But then it is impossible that
// the singularity has two branches, since mu is even and delta is an
// integer!
// So obviously, we did not decompose completely. Because the second branch
// is smooth, only the first ideal can be the one which can be decomposed
// further.
// Let us now consider the normalization of this first ideal i1.
LIB "normal.lib";
normal(i2);
↦
↦ // 'normal' created a list, say nor, of two elements.
↦ // To see the list type
↦       nor;
↦
↦ // * nor[1] is a list of 1 ring(s).
↦ // To access the i-th ring nor[1][i], give it a name, say Ri, and type
↦      def R1 = nor[1][1]; setring R1; norid; normap;
↦ // For the other rings type first (if R is the name of your base ring)
↦      setring R;
↦ // and then continue as for R1.
↦ // Ri/norid is the affine algebra of the normalization of R/P_i where
↦ // P_i is the i-th component of a decomposition of the input ideal id
↦ // and normap the normalization map from R to Ri/norid.
↦
↦ // * nor[2] is a list of 1 ideal(s). Let ci be the last generator
```

```
↦ // of the ideal nor[2][i]. Then the integral closure of R/P_i is
↦ // generated as R-submodule of the total ring of fractions by
↦ // 1/ci * nor[2][i].
↦ [1]:
↦    [1]:
↦       // coefficients: QQ
↦ // number of vars : 6
↦ //        block   1 : ordering dp
↦ //                  : names    T(1) T(2) T(3)
↦ //        block   2 : ordering dp
↦ //                  : names    x y z
↦ //        block   3 : ordering C
↦ [2]:
↦    [1]:
↦       _[1]=y
↦       _[2]=xz
↦       _[3]=x2
↦       _[4]=z2
  def rno=_[1][1];
  setring rno;
  norid;
↦ norid[1]=-T(2)*z+x
↦ norid[2]=T(1)*x-z
↦ norid[3]=T(2)*x-T(3)*z
↦ norid[4]=T(1)*z+T(2)*z+T(3)*x+T(3)*z+z
↦ norid[5]=-T(2)*y+z^2
↦ norid[6]=T(1)*z^2-y
↦ norid[7]=T(2)*z^2-T(3)*y
↦ norid[8]=T(1)*y+T(2)*y+T(3)*z^2+T(3)*y+y
↦ norid[9]=T(1)^2+T(1)+T(2)+T(3)+1
↦ norid[10]=T(1)*T(2)-1
↦ norid[11]=T(2)^2-T(3)
↦ norid[12]=T(1)*T(3)-T(2)
↦ norid[13]=T(2)*T(3)+T(1)+T(2)+T(3)+1
↦ norid[14]=T(3)^2-T(1)
↦ norid[15]=z^3-x*y
↦ norid[16]=x^3+x^2*z+x*z^2+x*y+y*z
↦ norid[17]=x^2*z^2+x^2*y+x*y*z+y*z^2+y^2
  // The ideal is generated by a polynomial in one variable of degree 4 which
  // factors completely into 4 polynomials of type T(2)+a.
  // From this, we know that the ring of the normalization is the direct sum of
  // 4 polynomial rings in one variable.
  // Hence our original curve has these 4 branches plus a smooth one
  // which we already determined by primary decomposition.
  // Our final result is therefore: 5 branches.
```

A.4.8 Classification of hypersurface singularities

Classification of isolated hypersurface singularities with respect to right equivalence is provided by the command `classify` of the library `classify.lib`. The classification is done by using the algorithm of Arnold. Before entering this algorithm, a first guess based on the Hilbert polynomial of the Milnor algebra is made.

```
  LIB "classify.lib";
```

```
  ring r=0,(x,y,z),ds;
  poly p=singularity("E[6k+2]",2)[1];
  p=p+z^2;
  p;
↦ z2+x3+xy6+y8
  // We received an E_14 singularity in normal form
  // from the database of normal forms. Since only the residual
  // part is saved in the database, we added z^2 to get an E_14
  // of embedding dimension 3.
  //
  // Now we apply a coordinate change in order to deal with a
  // singularity which is not in normal form:
  map phi=r,x+y,y+z,x;
  poly q=phi(p);
  // Yes, q really looks ugly, now:
  q;
↦ x2+x3+3x2y+3xy2+y3+xy6+y7+6xy5z+6y6z+15xy4z2+15y5z2+20xy3z3+20y4z3+15xy2z\
   4+15y3z4+6xyz5+6y2z5+xz6+yz6+y8+8y7z+28y6z2+56y5z3+70y4z4+56y3z5+28y2z6+8\
   yz7+z8
  // Classification
  classify(q);
↦ About the singularity :
↦           Milnor number(f)   = 14
↦           Corank(f)          = 2
↦           Determinacy       <= 12
↦ Guessing type via Milnorcode:   E[6k+2]=E[14]
↦ 
↦ Computing normal form ...
↦ I have to apply the splitting lemma. This will take some time....:-)
↦    Arnold step number 9
↦ The singularity
↦    x3-9/4x4+27/4x5-189/8x6+737/8x7+6x6y+15x5y2+20x4y3+15x3y4+6x2y5+xy6-24\
  089/64x8-x7y+11/2x6y2+26x5y3+95/2x4y4+47x3y5+53/2x2y6+8xy7+y8+104535/64x9\
  +27x8y+135/2x7y2+90x6y3+135/2x5y4+27x4y5+9/2x3y6-940383/128x10-405/4x9y-2\
  025/8x8y2-675/2x7y3-2025/8x6y4-405/4x5y5-135/8x4y6+4359015/128x11+1701/4x\
  10y+8505/8x9y2+2835/2x8y3+8505/8x7y4+1701/4x6y5+567/8x5y6-82812341/512x12\
  -15333/8x11y-76809/16x10y2-25735/4x9y3-78525/16x8y4-16893/8x7y5-8799/16x6\
  y6-198x5y7-495/4x4y8-55x3y9-33/2x2y10-3xy11-1/4y12
↦ is R-equivalent to E[14].
↦    Milnor number = 14
↦    modality      = 1
↦ 2z2+x3+xy6+y8
  // The library also provides routines to determine the corank of q
  // and its residual part without going through the whole
  // classification algorithm.
  corank(q);
↦ 2
  morsesplit(q);
↦ y3-9/4y4+27/4y5-189/8y6+737/8y7+6y6z+15y5z2+20y4z3+15y3z4+6y2z5+yz6-24089\
  /64y8-y7z+11/2y6z2+26y5z3+95/2y4z4+47y3z5+53/2y2z6+8yz7+z8+104535/64y9+27\
  y8z+135/2y7z2+90y6z3+135/2y5z4+27y4z5+9/2y3z6-940383/128y10-405/4y9z-2025\
  /8y8z2-675/2y7z3-2025/8y6z4-405/4y5z5-135/8y4z6+4359015/128y11+1701/4y10z\
  +8505/8y9z2+2835/2y8z3+8505/8y7z4+1701/4y6z5+567/8y5z6-82812341/512y12-15\
```

```
333/8y11z-76809/16y10z2-25735/4y9z3-78525/16y8z4-16893/8y7z5-8799/16y6z6-\
198y5z7-495/4y4z8-55y3z9-33/2y2z10-3yz11-1/4z12
```

A.4.9 Resolution of singularities

Resolution of singularities and applications thereof are provided by the libraries `resolve.lib` and `reszeta.lib`; graphical output may be generated automatically by using external programs `surf` and `dot` respectively to which a specialized interface is provided by the library `resgraph.lib`. In this example, the basic functionality of the resolution of singularities package is illustrated by the computation of the intersection matrix and genera of the exceptional curves on a surface obtained from resolving the A6 surface singularity. A separate tutorial, which introduces the complete functionality of the package and explains the rather complicated data structures appearing in intermediate results, can be found at `http://www.singular.uni-kl.de/tutor_resol.ps`.

```
LIB"resolve.lib";                     // load the resolution algorithm
LIB"reszeta.lib";                     // load its application algorithms

ring R=0,(x,y,z),dp;                // define the ring Q[x,y,z]
ideal I=x7+y2-z2;                   // an A6 surface singularity
list L=resolve(I);                  // compute the resolution
list iD=intersectionDiv(L);         // compute intersection properties
iD;                                 // show the output
↦ [1]:
↦    -2,0,1,0,0,0,
↦    0,-2,0,1,0,0,
↦    1,0,-2,0,1,0,
↦    0,1,0,-2,0,1,
↦    0,0,1,0,-2,1,
↦    0,0,0,1,1,-2
↦ [2]:
↦    0,0,0,0,0,0
↦ [3]:
↦    [1]:
↦       [1]:
↦          2,1,1
↦       [2]:
↦          4,1,1
↦    [2]:
↦       [1]:
↦          2,1,2
↦       [2]:
↦          4,1,2
↦    [3]:
↦       [1]:
↦          4,2,1
↦       [2]:
↦          6,2,1
↦    [4]:
↦       [1]:
↦          4,2,2
↦       [2]:
↦          6,2,2
↦    [5]:
↦       [1]:
```

```
↦          6,3,1
↦       [2]:
↦          7,3,1
↦    [6]:
↦       [1]:
↦          6,3,2
↦       [2]:
↦          7,3,2
↦ [4]:
↦    1,1,1,1,1,1
// The output is a list whose first entry contains the intersection matrix
// of the exceptional divisors. The second entry is the list of genera
// of these divisors. The third and fourth entry contain the information
// how to find the corresponding divisors in the respective charts.
```

A.5 Invariant Theory

A.5.1 G_a -Invariants

We work in characteristic 0 and use the Lie algebra generated by one vectorfield of the form $\sum x_i \partial/\partial x_{i+1}$.

```
  LIB "ainvar.lib";
  int n=5;
  int i;
  ring s=32003,(x(1..n)),wp(1,2,3,4,5);
  // definition of the vectorfield m=sum m[i,1]*d/dx(i)
  matrix m[n][1];
  for (i=1;i<=n-1;i=i+1)
  {
     m[i+1,1]=x(i);
  }
  // computation of the ring of invariants
  ideal in=invariantRing(m,x(2),x(1),0);
  in;   //invariant ring is generated by 5 invariants
↦ in[1]=x(1)
↦ in[2]=x(2)^2-2*x(1)*x(3)
↦ in[3]=x(3)^2-2*x(2)*x(4)+2*x(1)*x(5)
↦ in[4]=x(2)^3-3*x(1)*x(2)*x(3)+3*x(1)^2*x(4)
↦ in[5]=x(3)^3-3*x(2)*x(3)*x(4)-15997*x(1)*x(4)^2+3*x(2)^2*x(5)-6*x(1)*x(3)\
   *x(5)
  ring q=32003,(x,y,z,u,v,w),dp;
  matrix m[6][1];
  m[2,1]=x;
  m[3,1]=y;
  m[5,1]=u;
  m[6,1]=v;
  // the vectorfield is: xd/dy+yd/dz+ud/dv+vd/dw
  ideal in=invariantRing(m,y,x,0);
  in; //invariant ring is generated by 6 invariants
↦ in[1]=x
↦ in[2]=u
↦ in[3]=v2-2uw
```

```
↦ in[4]=zu-yv+xw
↦ in[5]=yu-xv
↦ in[6]=y2-2xz
```

A.5.2 Invariants of a finite group

Two algorithms to compute the invariant ring are implemented in SINGULAR, `invariant_ring` and `invariant_ring_random`, both by Agnes E. Heydtmann (`agnes@math.uni-sb.de`).

Bases of homogeneous invariants are generated successively and those are chosen as primary invariants that lower the dimension of the ideal generated by the previously found invariants (see paper "Generating a Noetherian Normalization of the Invariant Ring of a Finite Group" by Decker, Heydtmann, Schreyer (J.Symb.Comput. 25, No.6, 727-731, 1998). In the non-modular case secondary invariants are calculated by finding a basis (in terms of monomials) of the basering modulo the primary invariants, mapping to invariants with the Reynolds operator and using those or their power products such that they are linearly independent modulo the primary invariants (see paper "Some Algorithms in Invariant Theory of Finite Groups" by Kemper and Steel (In: Proceedings of the Euroconference in Essen 1997, Birkhäuser Prog. Math. 173, 267-285, 1999)). In the modular case they are generated according to "Calculating Invariant Rings of Finite Groups over Arbitrary Fields" by Kemper (J.Symb.Comput. 21, No.3, 351-366, 1996).

We calculate now an example from Sturmfels: "Algorithms in Invariant Theory 2.3.7":

```
  LIB "finvar.lib";
  ring R=0,(x,y,z),dp;
  matrix A[3][3]=0,1,0,-1,0,0,0,0,-1;
  // the group G is generated by A in Gl(3,Q);
  print(A);
↦ 0, 1,0,
↦ -1,0,0,
↦ 0, 0,-1
  print(A*A*A*A); // the fourth power of A is 1
↦ 1,0,0,
↦ 0,1,0,
↦ 0,0,1
  // Use the first method to compute the invariants of G:
  matrix B(1..3);
  B(1..3)=invariant_ring(A);
  // SINGULAR returns 2 matrices, the first containing
  // primary invariants and the second secondary
  // invariants, i.e., module generators over a Noetherian
  // normalization
  // the third result are the irreducible secondary invariants
  // if the Molien series was available
  print(B(1));
↦ z2,x2+y2,x2y2
  print(B(2));
↦ 1,xyz,x2z-y2z,x3y-xy3
  print(B(3));
↦ xyz,x2z-y2z,x3y-xy3
  // Use the second method,
  // with random numbers between -1 and 1:
  B(1..3)=invariant_ring_random(A,1);
  print(B(1..3));
↦ z2,x2+y2,x4+y4-z4
```

```
↦  1,xyz,x2z-y2z,x3y-xy3
↦  xyz,x2z-y2z,x3y-xy3
```

A.6 Geometric Invariant Theory

A.6.1 GIT-Fans

Dolgachev/Hu and Thaddeus assigned to an algebraic variety with the action of an algebraic group the GIT-fan, a polyhedral fan enumerating the GIT-quotients in the sense of Mumford. The case of the action of an algebraic torus H on an affine variety X has been treated by Berchtold/Hausen. Based on their construction, an algorithm to compute the GIT-fan in this setting has been proposed be Keicher. Note that this setting is essential for many applications, since the torus case can be used to investigate the GIT-variation of the action of a connected reductive group G. In many important examples, X is symmetric under the action of a fnite group which either is known directly from its geometry or can be computed. A prominent instance is the Deligne-Mumford compactification M06bar of the moduli space of 6-pointed stable curves of genus zero, which has a natural action of the symmetric group S6. The library gitfan.lib implements an efficient algorithm for computing GIT-fans, which makes use of symmetries. We have applied this algorithm to determine the Mori chamber decomposition of the cone of movable divisor classes of M06bar. Each cone is represented by a single integer. The algorithm relies on Groebner basis techniques, convex geometry and actions of finite symmetry groups. It demonstrates the strength of cross-boarder methods in computer algebra, and the efficiency of the algorithms implemented in all involved areas. The algorithm is also suitable for parallel computations.

As an example we address in the following the computation of the GIT-Fan of M05bar.

We first compute the GIT-fan using the single line command provided by the library:

```
LIB "gitfan.lib";
setcores(4);
ring R = 0,T(1..10),wp(1,1,1,1,1,1,1,1,1,1);
ideal J =
  T(5)*T(10)-T(6)*T(9)+T(7)*T(8),
  T(1)*T(9)-T(2)*T(7)+T(4)*T(5),
  T(1)*T(8)-T(2)*T(6)+T(3)*T(5),
  T(1)*T(10)-T(3)*T(7)+T(4)*T(6),
  T(2)*T(10)-T(3)*T(9)+T(4)*T(8);
intmat Q[5][10] =
  1, 1, 1, 1, 0, 0, 0, 0, 0, 0,
  1, 0, 0, 0, 1, 1, 1, 0, 0, 0,
  0, 1, 1, 0, 0, 0, -1, 1, 0, 0,
  0, 1, 0, 1, 0, -1, 0, 0, 1, 0,
  0, 0, 1, 1, -1, 0, 0, 0, 0, 1;
fan GIT = GITfan(J,Q);
fVector(GIT);
```

The GIT-Fan can be computed using symmetries as follows:

```
LIB "gitfan.lib";
setcores(4);
ring R = 0,T(1..10),wp(1,1,1,1,1,1,1,1,1,1);
ideal J =
  T(5)*T(10)-T(6)*T(9)+T(7)*T(8),
  T(1)*T(9)-T(2)*T(7)+T(4)*T(5),
  T(1)*T(8)-T(2)*T(6)+T(3)*T(5),
```

```
    T(1)*T(10)-T(3)*T(7)+T(4)*T(6),
    T(2)*T(10)-T(3)*T(9)+T(4)*T(8);
  intmat Q[5][10] =
    1, 1, 1, 1, 0, 0, 0, 0, 0, 0,
    1, 0, 0, 0, 1, 1, 1, 0, 0, 0,
    0, 1, 1, 0, 0, 0, -1, 1, 0, 0,
    0, 1, 0, 1, 0, -1, 0, 0, 1, 0,
    0, 0, 1, 1, -1, 0, 0, 0, 0, 1;
  list simplexSymmetryGroup = G25Action();
  fan GIT2 = GITfan(J,Q,simplexSymmetryGroup);
  GIT2;
```

Although we provide a procedure to compute the orbit decomposition of the group action on the simplex of variables this is not fast in Singular. In the following we describe how to use GAP to obtain the orbit decomposition and then continue with this data in Singular. This is particularily useful for more complicated examples.

The file orbits.gp in the directory doc of the Singular source tree contains GAP code to do this computation. This result is provided in the file doc/simplexOrbitRepresentativesG25.sing.

The file doc/simplexSymmetryGroupG25.sing contains the symmetry group (which here is S5).

Moreover the file doc/elementsInTermsOfGeneratorsG25.sing contains a representation of the elements of the symmetry group in terms of generators.

```
LIB "gitfan.lib";
setcores(4);
ring R = 0,T(1..10),wp(1,1,1,1,1,1,1,1,1,1);
ideal J =
  T(5)*T(10)-T(6)*T(9)+T(7)*T(8),
  T(1)*T(9)-T(2)*T(7)+T(4)*T(5),
  T(1)*T(8)-T(2)*T(6)+T(3)*T(5),
  T(1)*T(10)-T(3)*T(7)+T(4)*T(6),
  T(2)*T(10)-T(3)*T(9)+T(4)*T(8);
intmat Q[5][10] =
  1, 1, 1, 1, 0, 0, 0, 0, 0, 0,
  1, 0, 0, 0, 1, 1, 1, 0, 0, 0,
  0, 1, 1, 0, 0, 0, -1, 1, 0, 0,
  0, 1, 0, 1, 0, -1, 0, 0, 1, 0,
  0, 0, 1, 1, -1, 0, 0, 0, 0, 1;
intmat Qt = transpose(Q);
<"doc/simplexOrbitRepresentativesG25.sing";
list afaceOrbitRepresentatives=afaces(J,simplexOrbitRepresentatives);
<"doc/simplexSymmetryGroupG25.sing";
list fulldimAfaceOrbitRepresentatives=fullDimImages(afaceOrbitRepresentatives,Q);
list afaceOrbits=computeAfaceOrbits(fulldimAfaceOrbitRepresentatives,simplexSymmetryGroup);
apply(afaceOrbits,size);
list minAfaceOrbits = minimalAfaceOrbits(afaceOrbits);
apply(minAfaceOrbits,size);
list listOfOrbitConeOrbits = orbitConeOrbits(minAfaceOrbits,Q);
apply(listOfOrbitConeOrbits,size);
list listOfMinimalOrbitConeOrbits = minimalOrbitConeOrbits(listOfOrbitConeOrbits);
size(listOfMinimalOrbitConeOrbits);
cone mov = coneViaPoints(transpose(Q));
mov = canonicalizeCone(mov);
list OC =  listOfOrbitConeOrbits;
```

```
<"doc/elementsInTermsOfGeneratorsG25.sing";
list Asigmagens = groupActionOnQImage(generatorsG,Q);
list actionOnOrbitconeIndicesForGenerators = groupActionOnHashes(Asigmagens,OC);
list actionOnOrbitconeIndices;
for (int i =1; i<=size(elementsInTermsOfGenerators);i++)
{
  actionOnOrbitconeIndices[i]=evaluateProduct(actionOnOrbitconeIndicesForGenerators, elementsInTermsOfGenerators[i]);
}
list OClist = OC[1];
for (i =2;i<=size(OC);i++)
{
OClist = OClist + OC[i];
}
list SigmaHashes = GITfanParallelSymmetric(OClist, Q, mov, actionOnOrbitconeIndices);
SigmaHashes;
fan Sigma = hashesToFan(SigmaHashes,OClist);
```

Note that the result is not the complete fan but only the fan generated by a minimal set of representatives of maximal cones for the group action (by the group generated by Asigmagens).

A.7 Non-commutative Algebra

A.7.1 Left and two-sided Groebner bases

For a set of polynomials (resp. vectors) `S` in a non-commutative G-algebra, Singular:Plural provides two algorithms for computing Groebner bases.

The command `std` computes a left Groebner basis of a left module, generated by the set `S` (see Section 7.3.26 [std (plural)], page 355). The command `twostd` computes a two-sided Groebner basis (which is in particular also a left Groebner basis) of a two-sided ideal, generated by the set `S` (see Section 7.3.29 [twostd], page 358).

In the example below, we consider a particular set `S` in the algebra $A := U(sl_2)$ with the degree reverse lexicographic ordering. We compute a left Groebner basis `L` of the left ideal generated by `S` and a two-sided Groebner basis `T` of the two-sided ideal generated by `S`.
Then, we read off the information on the vector space dimension of the factor modules `A/L` and `A/T` using the command `vdim` (see Section 7.3.30 [vdim (plural)], page 358).
Further on, we use the command `reduce` (see Section 7.3.23 [reduce (plural)], page 350) to compare the left ideals generated by `L` and `T`.

We set `option(redSB)` and `option(redTail)` to make Singular compute completely reduced minimal bases of ideals (see Section 5.1.108 [option], page 230 and Section 7.4.2 [Groebner bases in G-algebras], page 360 for definitions and further details).

For long running computations, it is always recommended to set `option(prot)` to make Singular display some information on the performed computations (see Section 5.1.108 [option], page 230 for an interpretation of the displayed symbols).

```
// ----- 1.  setting up the algebra
  ring R = 0,(e,f,h),dp;
  matrix D[3][3];
  D[1,2]=-h; D[1,3]=2*e; D[2,3]=-2*f;
  def A=nc_algebra(1,D); setring A;
// ----- equivalently, you may use the following:
//  LIB "ncalg.lib";
```

```
//  def A = makeUsl2();
//  setring A;
// ----- 2.  defining the set S
  ideal S = e^3, f^3, h^3 - 4*h;
  option(redSB);
  option(redTail);
  option(prot);  // let us activate the protocol
  ideal L = std(S);
↦ 3(2)s
↦ s
↦ s
↦ 5s
↦ s
↦ (4)s
↦ 4(5)(4)s
↦ (6)(5)(4)s
↦ 3(7)4(5)(4)(3)s
↦ 3(4)(3)4(2)s
↦ (3)(2)s
↦ 3(5)(4)4(2)5
↦ (S:5)-----
↦ product criterion:7 chain criterion:12
  L;
↦ L[1]=h3-4h
↦ L[2]=fh2-2fh
↦ L[3]=eh2+2eh
↦ L[4]=2efh-h2-2h
↦ L[5]=f3
↦ L[6]=e3
  vdim(L); // the vector space dimension of the module A/L
↦ 15
  option(noprot); // turn off the protocol
  ideal T = twostd(S);
  T;
↦ T[1]=h3-4h
↦ T[2]=fh2-2fh
↦ T[3]=eh2+2eh
↦ T[4]=f2h-2f2
↦ T[5]=2efh-h2-2h
↦ T[6]=e2h+2e2
↦ T[7]=f3
↦ T[8]=ef2-fh
↦ T[9]=e2f-eh-2e
↦ T[10]=e3
  vdim(T);  // the vector space dimension of the module A/T
↦ 10
  print(matrix(reduce(L,T)));  // reduce L with respect to T
↦ 0,0,0,0,0,0
  // as we see, L is included in the left ideal generated by T
  print(matrix(reduce(T,L)));  // reduce T with respect to L
↦ 0,0,0,f2h-2f2,0,e2h+2e2,0,ef2-fh,e2f-eh-2e,0
  // the non-zero elements belong to T only
  ideal LT = twostd(L); // the two-sided Groebner basis of L
```

```
  // LT and T coincide as left ideals:
  size(reduce(LT,T));
↦ 0
  size(reduce(T,LT));
↦ 0
```

A.7.2 Right Groebner bases and syzygies

Most of the SINGULAR:PLURAL commands correspond to the *left-sided* computations, that is left Groebner bases, left syzygies, left resolutions and so on. However, the *right-sided* computations can be done, using the *left-sided* functionality and *opposite* algebras.

In the example below, we consider the algebra $A := U(sl_2)$ and a set of generators $I = \{e^2, f\}$.

We will compute a left Groebner basis LI and a left syzygy module LS of a left ideal, generated by the set I .

Then, we define the opposite algebra Aop of A, set it as a basering, and create opposite objects of already computed ones.

Further on, we compute a right Groebner basis RI and a right syzygy module RS of a right ideal, generated by the set I in A .

```
// ----- setting up the algebra:
LIB "ncalg.lib";
def A = makeUsl2();
setring A; A;
↦ // coefficients: QQ
↦ // number of vars : 3
↦ //        block   1 : ordering dp
↦ //                  : names    e f h
↦ //        block   2 : ordering C
↦ // noncommutative relations:
↦ //    fe=ef-h
↦ //    he=eh+2e
↦ //    hf=fh-2f
// ----- equivalently, you may use
// ring AA = 0,(e,f,h),dp;
// matrix D[3][3];
// D[1,2]=-h; D[1,3]=2*e; D[2,3]=-2*f;
// def A=nc_algebra(1,D); setring A;
option(redSB);
option(redTail);
matrix T;
// --- define a generating set
ideal   I = e2,f;
ideal  LI = std(I); // the left Groebner basis of I
LI;             // we see that I was not a Groebner basis
↦ LI[1]=f
↦ LI[2]=h2+h
↦ LI[3]=eh+e
↦ LI[4]=e2
module LS = syz(I); // the left syzygy module of I
print(LS);
↦ -ef-2h+6,-f3,                    -ef2-fh+4f,  -e2f2-4efh+16ef-6h2+42h-72\
   ,
```

```
↦ e3,      e2f2-6efh-6ef+6h2+18h+12,e3f-3e2h-6e2,e4f
 // check: LS is a left syzygy, if T=0:
T  = transpose(LS)*transpose(I);
print(T);
↦ 0,
↦ 0,
↦ 0,
↦ 0
// --- let us define the opposite algebra of A
def Aop = opposite(A);
setring Aop; Aop;         // see how Aop looks like
↦ // coefficients: QQ
↦ // number of vars : 3
↦ //        block   1 : ordering a
↦ //                  : names    H F E
↦ //                  : weights  1 1 1
↦ //        block   2 : ordering ls
↦ //                  : names    H F E
↦ //        block   3 : ordering C
↦ // noncommutative relations:
↦ //    FH=HF-2F
↦ //    EH=HE+2E
↦ //    EF=FE-H
// --- we "oppose" (transfer) objects from A to Aop
ideal   Iop = oppose(A,I);
ideal  RIop = std(Iop);  // the left Groebner basis of Iop in Aop
module RSop = syz(Iop);  // the left syzygy module of Iop in Aop
module LSop = oppose(A,LS);
module RLS  = syz(transpose(LSop));
// RLS is the left syzygy of transposed LSop in Aop
// --- let us return to A and transfer (i.e. oppose)
// all the computed objects back
setring A;
ideal  RI = oppose(Aop,RIop); // the right Groebner basis of I
RI;              // it differs from the left Groebner basis LI
↦ RI[1]=f
↦ RI[2]=h2-h
↦ RI[3]=eh+e
↦ RI[4]=e2
module RS = oppose(Aop,RSop); // the right syzygy module of I
print(RS);
↦ -ef+3h+6,-f3,                     -ef2+3fh,-e2f2+4efh+4ef,
↦ e3,      e2f2+2efh-6ef+2h2-10h+12,e3f,     e4f
 // check: RS is a right syzygy, if T=0:
T = matrix(I)*RS;
T;
↦ T[1,1]=0
↦ T[1,2]=0
↦ T[1,3]=0
↦ T[1,4]=0
module RLS;
RLS = transpose(oppose(Aop,RLS));
// RLS is the right syzygy of a left syzygy of I
```

```
// it is I itself ?
print(RLS);
↦ e2,f
```

A.8 Applications

A.8.1 Solving systems of polynomial equations

Here we turn our attention to the probably most popular aspect of the solving problem: given a system of complex polynomial equations with only finitely many solutions, compute floating point approximations for these solutions. This is widely considered as a task for numerical analysis. However, due to rounding errors, purely numerical methods are often unstable in an unpredictable way.

Therefore, in many cases, it is worth investing more computing power to derive additional knowledge on the geometric structure of the set of solutions (not to mention the question of how to decide whether the set of solutions is finite or not). The symbolic-numerical approach to the solving problem combines numerical methods with a symbolic preprocessing.

Depending on whether we want to preserve the multiplicities of the solutions or not, possible goals for a symbolic preprocessing are

- to find another system of generators (for instance, a reduced Groebner basis) for the ideal I generated by the polynomial equations. Alternatively, find a system of polynomials defining an ideal which has the same radical as I (see Section A.2 [Computing Groebner and Standard Bases], page 638, resp. [radical], page 757).

In any case, the goal should be to find a system for which a numerical solution can be found more easily and in a more stable way. For systems with a large number of generators, the first step in a Singular computation could be to reduce the number of generators by applying the `interred` command (see Section 5.1.62 [interred], page 199). Another goal might be

- to decompose the system into several smaller (or, at least, more accessible) systems of polynomial equations. Then, the set of solutions of the original system is obtained by taking the union of the sets of solutions of the new systems.

Such a decomposition can be obtained in several ways: for instance, by computing a triangular decomposition (see Section D.8.4 [triang_lib], page 792) for the ideal I, or by applying the factorizing Buchberger algorithm (see Section 5.1.33 [facstd], page 176), or by computing a primary decomposition of I (see Section D.4.21 [primdec_lib], page 756).

Moreover, the equational modelling of a problem frequently causes unwanted solutions, for instance, zero as a multiple solution. Not only for stability reasons, one is frequently interested to get rid of those. This can be done by computing the saturation of I with respect to an ideal having the excess components as set of solutions (see [sat], page 743).

The Singular libraries `solve.lib` and `triang.lib` provide several commands for solving systems of polynomial equations (based on a symbolic-numerical approach via Groebner bases, resp. resultants). In the example below, we show some of these commands at work.

```
LIB "solve.lib";
ring r=0,x(1..5),dp;
poly f0= x(1)^3+x(2)^2+x(3)^2+x(4)^2-x(5)^2;
poly f1= x(2)^3+x(1)^2+x(3)^2+x(4)^2-x(5)^2;
poly f2=x(3)^3+x(1)^2+x(2)^2+x(4)^2-x(5)^2;
poly f3=x(4)^2+x(1)^2+x(2)^2+x(3)^2-x(5)^2;
poly f4=x(5)^2+x(1)^2+x(2)^2+x(3)^2;
```

```
ideal i=f0,f1,f2,f3,f4;
ideal si=std(i);
//
// dimension of a solution set (here: 0) can be read from a Groebner bases
// (with respect to any global monomial ordering)
dim(si);
↦ 0
//
// the number of complex solutions (counted with multiplicities) is:
vdim(si);
↦ 108
//
// The given system has a multiple solution at the origin. We use facstd
// to compute equations for the non-zero solutions:
option(redSB);
ideal maxI=maxideal(1);
ideal j=sat(si,maxI)[1];  // output is Groebner basis
vdim(j);                   // number of non-zero solutions (with mult's)
↦ 76
//
// We compute a triangular decomposition for the ideal I. This requires first
// the computation of a lexicographic Groebner basis (we use the FGLM
// conversion algorithm):
ring R=0,x(1..5),lp;
ideal j=fglm(r,j);
list L=triangMH(j);
size(L);                  // number of triangular components
↦ 7
L[1];                     // the first component
↦ _[1]=x(5)^2+1
↦ _[2]=x(4)^2+2
↦ _[3]=x(3)-1
↦ _[4]=x(2)^2
↦ _[5]=x(1)^2
//
// We compute floating point approximations for the solutions (with 30 digits)
def S=triang_solve(L,30);
↦
↦ // 'triang_solve' created a ring, in which a list rlist of numbers (the
↦ // complex solutions) is stored.
↦ // To access the list of complex solutions, type (if the name R was assig\
   ned
↦ // to the return value):
↦         setring R; rlist;
setring S;
size(rlist);              // number of different non-zero solutions
↦ 28
rlist[1];                 // the first solution
↦ [1]:
↦ 0
↦ [2]:
↦ 0
↦ [3]:
```

```
↦ 1
↦ [4]:
↦ (-I*1.41421356237309504880168872421)
↦ [5]:
↦ -I
//
// Alternatively, we could have applied directly the solve command:
setring r;
def T=solve(i,30,1,"nodisplay");  // compute all solutions with mult's
↦
↦ // 'solve' created a ring, in which a list SOL of numbers (the complex so\
   lutions)
↦ // is stored.
↦ // To access the list of complex solutions, type (if the name R was assig\
   ned
↦ // to the return value):
↦         setring R; SOL;
setring T;
size(SOL);               // number of different solutions
↦ 4
SOL[1][1]; SOL[1][2];    // first solution and its multiplicity
↦ [1]:
↦    [1]:
↦ 1
↦    [2]:
↦ 1
↦    [3]:
↦ 1
↦    [4]:
↦ (i*2.44948974278317809819728407471)
↦    [5]:
↦ (i*1.73205080756887729352744634151)
↦ [2]:
↦    [1]:
↦ 1
↦    [2]:
↦ 1
↦    [3]:
↦ 1
↦    [4]:
↦ (-i*2.44948974278317809819728407471)
↦    [5]:
↦ (i*1.73205080756887729352744634151)
↦ [3]:
↦    [1]:
↦ 1
↦    [2]:
↦ 1
↦    [3]:
↦ 1
↦    [4]:
↦ (i*2.44948974278317809819728407471)
↦    [5]:
```

```
↦ (-i*1.73205080756887729352744634151)
↦ [4]:
↦    [1]:
↦ 1
↦    [2]:
↦ 1
↦    [3]:
↦ 1
↦    [4]:
↦ (-i*2.44948974278317809819728407471)
↦    [5]:
↦ (-i*1.73205080756887729352744634151)
↦ 1
SOL[size(SOL)];          // solutions of highest multiplicity
↦ [1]:
↦    [1]:
↦       [1]:
↦ 0
↦       [2]:
↦ 0
↦       [3]:
↦ 0
↦       [4]:
↦ 0
↦       [5]:
↦ 0
↦ [2]:
↦    32
//
// Or, we could remove the multiplicities first, by computing the
// radical:
setring r;
ideal k=std(radical(i));
vdim(k);                 // number of different complex solutions
↦ 29
def T1=solve(k,30,"nodisplay");  // compute all solutions with mult's
↦
↦ // 'solve' created a ring, in which a list SOL of numbers (the complex so\
   lutions)
↦ // is stored.
↦ // To access the list of complex solutions, type (if the name R was assig\
   ned
↦ // to the return value):
↦         setring R; SOL;
setring T1;
size(SOL);               // number of different solutions
↦ 29
SOL[1];
↦ [1]:
↦ 1
↦ [2]:
↦ 1
↦ [3]:
```

```
↦ 1
↦ [4]:
↦ (-i*2.44948974278317809819728407471)
↦ [5]:
↦ (-i*1.73205080756887729352744634151)
```

A.8.2 AG codes

The library `brnoeth.lib` provides an implementation of the Brill-Noether algorithm for solving the Riemann-Roch problem and applications to Algebraic Geometry codes. The procedures can be applied to plane (singular) curves defined over a prime field of positive characteristic.

```
LIB "brnoeth.lib";
ring s=2,(x,y),lp;            // characteristic 2
poly f=x3y+y3+x;              // the Klein quartic
list KLEIN=Adj_div(f);        // compute the conductor
↦ Computing affine singular points ...
↦ Computing all points at infinity ...
↦ Computing affine singular places ...
↦ Computing singular places at infinity ...
↦ Computing non-singular places at infinity ...
↦ Adjunction divisor computed successfully
↦
↦ The genus of the curve is 3
KLEIN=NSplaces(1..3,KLEIN);   // computes places up to degree 3
↦ Computing non-singular affine places of degree 1 ...
↦ Computing non-singular affine places of degree 2 ...
↦ Computing non-singular affine places of degree 3 ...
KLEIN=extcurve(3,KLEIN);      // construct Klein quartic over F_8
↦
↦ Total number of rational places : NrRatPl = 24
↦
KLEIN[3];                     // display places (degree, number)
↦ [1]:
↦    1,1
↦ [2]:
↦    1,2
↦ [3]:
↦    1,3
↦ [4]:
↦    2,1
↦ [5]:
↦    3,1
↦ [6]:
↦    3,2
↦ [7]:
↦    3,3
↦ [8]:
↦    3,4
↦ [9]:
↦    3,5
↦ [10]:
↦    3,6
↦ [11]:
```

```
↦    3,7
// We define a divisor G of degree 14=6*1+4*2:
intvec G=6,0,0,4,0,0,0,0,0,0,0;   // 6 * place #1 + 4 * place #4
// We compute an evaluation code which evaluates at all rational places
// outside the support of G (place #4 is not rational)
intvec D=2..24;
// in D, the number i refers to the i-th element of the list POINTS in
// the ring KLEIN[1][5].
def RR=KLEIN[1][5];
setring RR; POINTS[1];        // the place in the support of G (not in supp(D))
↦ [1]:
↦ 0
↦ [2]:
↦ 1
↦ [3]:
↦ 0
setring s;
def RR=KLEIN[1][4];
↦ // ** redefining RR (def RR=KLEIN[1][4];)
setring RR;
matrix C=AGcode_L(G,D,KLEIN); // generator matrix for the evaluation AG code
↦ Forms of degree 5 :
↦ 21
↦
↦ Vector basis successfully computed
↦
nrows(C);
↦ 12
ncols(C);
↦ 23
//
// We can also compute a generator matrix for the residual AG code
matrix CO=AGcode_Omega(G,D,KLEIN);
↦ Forms of degree 5 :
↦ 21
↦
↦ Vector basis successfully computed
↦
//
// Preparation for decoding:
// We need a divisor of degree at least 6 whose support is disjoint with the
// support of D:
intvec F=6;                   // F = 6*point #1
// in F, the i-th entry refers to the i-th element of the list POINTS in
// the ring KLEIN[1][5]
list K=prepSV(G,D,F,KLEIN);
↦ Forms of degree 5 :
↦ 21
↦
↦ Vector basis successfully computed
↦
↦ Forms of degree 4 :
↦ 15
```

```
↦
↦ Vector basis successfully computed
↦
↦ Forms of degree 4 :
↦ 15
↦
↦ Vector basis successfully computed
↦
K[size(K)][1];                // error-correcting capacity
↦ 3
//
//  Encoding and Decoding:
matrix word[1][11];           // a word of length 11 is encoded
word = 1,1,1,1,1,1,1,1,1,1,1;
def y=word*C0;                // the code word (length: 23)
matrix disturb[1][23];
disturb[1,1]=1;
disturb[1,10]=a;
disturb[1,12]=1+a;
y=y+disturb;                  // disturb the code word (3 errors)
def yy=decodeSV(y,K);         // error correction
yy-y;                         // display the error
↦ _[1,1]=1
↦ _[1,2]=0
↦ _[1,3]=0
↦ _[1,4]=0
↦ _[1,5]=0
↦ _[1,6]=0
↦ _[1,7]=0
↦ _[1,8]=0
↦ _[1,9]=0
↦ _[1,10]=(a)
↦ _[1,11]=0
↦ _[1,12]=(a+1)
↦ _[1,13]=0
↦ _[1,14]=0
↦ _[1,15]=0
↦ _[1,16]=0
↦ _[1,17]=0
↦ _[1,18]=0
↦ _[1,19]=0
↦ _[1,20]=0
↦ _[1,21]=0
↦ _[1,22]=0
↦ _[1,23]=0
```

Appendix B Polynomial data

B.1 Representation of mathematical objects

SINGULAR distinguishes between objects which do not belong to a ring and those which belong to a specific ring (see Section 3.3 [Rings and orderings], page 30). We comment only on the latter ones.

Internally all ring-dependent objects are polynomials or structures built from polynomials (and some additional information). Note that SINGULAR stores (and hence prints) a polynomial automatically w.r.t. the monomial ordering.

The definition of ideals and matrices, respectively, is straight forward: The user gives a list of polynomials which generate the ideal, resp. which are the entries of the matrix. (The number of rows and columns need to be provided when creating the matrix.)

A vector in SINGULAR is always an element of a free module over the basering. It is given as a list of polynomials in one of the following formats $[f_1, ..., f_n]$ or $f_1 * gen(1) + ... + f_n * gen(n)$, where $gen(i)$ denotes the i-th canonical generator of a free module (with 1 at index i and 0 everywhere else). Both forms are equivalent. A vector is internally represented in the second form with the $gen(i)$ being "special" ring variables, ordered accordingly to the monomial ordering. Therefore, the form $[f_1, ..., f_n]$ serves as output only if the monomial ordering gives priority to the component, i.e., is of the form `(c,...)` (see Section B.2.5 [Module orderings], page 697). However, in any case the procedure `show` from the library `inout.lib` displays the bracket format.

A vector $v = [f_1, ..., f_n]$ should always be considered as a column vector in a free module of rank equal to nrows(v) where nrows(v) is equal to the maximal index r such that $f_r \neq 0$. This is due to the fact, that internally v is a polynomial in a sparse representation, i.e., $f_i * gen(i)$ is not stored if $f_i = 0$ (for reasons of efficiency), hence the last 0-entries of v are lost. Only more complex structures are able to keep the rank.

A module M in SINGULAR is given by a list of vectors $v_1, ..., v_k$ which generate the module as a submodule of the free module of rank equal to nrows(M) which is the maximum of nrows(v_i).

If one wants to create a module with a larger rank than given by its generators, one has to use the command `attrib(M,"rank",r)` (see Section 5.1.2 [attrib], page 154, Section 5.1.104 [nrows], page 229) or to define a matrix first, then converting it into a module. Modules in SINGULAR are almost the same as matrices, they may be considered as sparse representations of matrices. A module of a matrix is generated by the columns of the matrix and a matrix of a module has as columns the generators of the module. These conversions preserve the rank and the number of generators, resp. the number of rows and columns.

By the above remarks it might appear that SINGULAR is only able to handle submodules of a free module. However, this is not true. SINGULAR can compute with any finitely generated module over the basering R. Such a module, say N, is not represented by its generators but by its (generators and) relations. This means that $N = R^n/M$ where n is the number of generators of N and $M \subseteq R^n$ is the module of relations. In other words, defining a module M as a submodule of a free module R^n can also be considered as the definition of $N = R^n/M$.

Note that most functions, when applied to a module M, really deal with M. However, there are some functions which deal with $N = R^n/M$ instead of M.

For example, `std(M)` computes a standard basis of M (and thus gives another representation of N as $N = R^n/\text{std}(M)$). However, `dim(M)`, resp. `vdim(M)`, return $\dim(R^n/M)$, resp. $\dim_k(R^n/M)$ (if M is given by a standard basis).

The function `syz(M)` returns the first syzygy module of M, i.e., the module of relations of the given generators of M which is equal to the second syzygy module of N. Refer to the description of each function in Section 5.1 [Functions], page 154 to get information which module the function deals with.

The numbering in `res` and other commands for computing resolutions refers to a resolution of $N = R^n/M$ (see [res], page 721; Section C.3 [Syzygies and resolutions], page 703).

It is possible to compute in any field which is a valid ground field in SINGULAR. For doing so, one has to define a ring with the desired ground field and at least one variable. The elements of the field are of type number, but may also be considered as polynomials (of degree 0). Large computations should be faster if the elements of the field are defined as numbers.

The above remarks do also apply to quotient rings. Polynomial data are stored internally in the same manner, the only difference is that this polynomial representation is in general not unique. `reduce(f,std(0))` computes a normal form of a polynomial f in a quotient ring (cf. Section 5.1.127 [reduce], page 246).

B.2 Monomial orderings

B.2.1 Introduction to orderings

SINGULAR offers a great variety of monomial orderings which provide an enormous functionality, if used diligently. However, this flexibility might also be confusing for the novice user. Therefore, we recommend to those not familiar with monomial orderings to generally use the ordering `dp` for computations in the polynomial ring $K[x_1,\ldots,x_n]$, resp. `ds` for computations in the localization $\mathrm{Loc}_{(x)}K[x_1,\ldots,x_n]$.

For inhomogenous input ideals, standard (resp. groebner) bases computations are generally faster with the orderings $\mathrm{Wp}(w_1,\ldots,w_n)$ (resp. $\mathrm{Ws}(w_1,\ldots,w_n)$) if the input is quasihomogenous w.r.t. the weights w_1, …, w_n of x_1, …, x_n.

If the output needs to be "triangular" (resp. "block-triangular"), the lexicographical ordering `lp` (resp. lexicographical block-orderings) need to be used. However, these orderings usually result in much less efficient computations.

B.2.2 General definitions for orderings

A monomial ordering (term ordering) on $K[x_1,\ldots,x_n]$ is a total ordering $<$ on the set of monomials (power products) $\{x^\alpha \mid \alpha \in \mathbf{N^n}\}$ which is compatible with the natural semigroup structure, i.e., $x^\alpha < x^\beta$ implies $x^\gamma x^\alpha < x^\gamma x^\beta$ for any $\gamma \in \mathbf{N^n}$. We do not require $<$ to be a wellordering. See the literature cited in Section C.9 [References], page 719.

It is known that any monomial ordering can be represented by a matrix M in $GL(n,R)$, but, of course, only integer coefficients are of relevance in practice.

Global orderings are wellorderings (i.e., $1 < x_i$ for each variable x_i), local orderings satisfy $1 > x_i$ for each variable. If some variables are ordered globally and others locally we call it a mixed ordering. Local or mixed orderings are not wellorderings.

Let K be the ground field, $x = (x_1,\ldots,x_n)$ the variables and $<$ a monomial ordering, then Loc $K[x]$ denotes the localization of $K[x]$ with respect to the multiplicatively closed set

$$\{1 + g \mid g = 0 \text{ or } g \in K[x]\backslash\{0\} \text{ and } L(g) < 1\}.$$

Here, $L(g)$ denotes the leading monomial of g, i.e., the biggest monomial of g with respect to $<$. The result of any computation which uses standard basis computations has to be interpreted in Loc $K[x]$.

Note that the definition of a ring includes the definition of its monomial ordering (see Section 3.3 [Rings and orderings], page 30). SINGULAR offers the monomial orderings described in the following sections.

B.2.3 Global orderings

For all these orderings, we have Loc $K[x] = K[x]$

lp: lexicographical ordering:
$x^\alpha < x^\beta \Leftrightarrow \exists\, 1 \leq i \leq n : \alpha_1 = \beta_1, \ldots, \alpha_{i-1} = \beta_{i-1}, \alpha_i < \beta_i$.

rp: reverse lexicographical ordering:
$x^\alpha < x^\beta \Leftrightarrow \exists\, 1 \leq i \leq n : \alpha_n = \beta_n, \ldots, \alpha_{i+1} = \beta_{i+1}, \alpha_i < \beta_i$.

dp: degree reverse lexicographical ordering:
let $\deg(x^\alpha) = \alpha_1 + \cdots + \alpha_n$, then $x^\alpha < x^\beta \Leftrightarrow \deg(x^\alpha) < \deg(x^\beta)$ or
$\deg(x^\alpha) = \deg(x^\beta)$ and $\exists\, 1 \leq i \leq n : \alpha_n = \beta_n, \ldots, \alpha_{i+1} = \beta_{i+1}, \alpha_i > \beta_i$.

Dp: degree lexicographical ordering:
let $\deg(x^\alpha) = \alpha_1 + \cdots + \alpha_n$, then $x^\alpha < x^\beta \Leftrightarrow \deg(x^\alpha) < \deg(x^\beta)$ or
$\deg(x^\alpha) = \deg(x^\beta)$ and $\exists\, 1 \leq i \leq n : \alpha_1 = \beta_1, \ldots, \alpha_{i-1} = \beta_{i-1}, \alpha_i < \beta_i$.

wp: weighted reverse lexicographical ordering:
let $w_1, \ldots, w_n$ be positive integers. Then `wp`$(w_1, \ldots, w_n)$ is defined as `dp` but with $\deg(x^\alpha) = w_1\alpha_1 + \cdots + w_n\alpha_n$.

Wp: weighted lexicographical ordering:
let $w_1, \ldots, w_n$ be positive integers. Then `Wp`$(w_1, \ldots, w_n)$ is defined as `Dp` but with $\deg(x^\alpha) = w_1\alpha_1 + \cdots + w_n\alpha_n$.

B.2.4 Local orderings

For ls, ds, Ds and, if the weights are positive integers, also for ws and Ws, we have Loc $K[x] = K[x]_{(x)}$, the localization of $K[x]$ at the maximal ideal $(x) = (x_1, ..., x_n)$.

ls: negative lexicographical ordering:
$x^\alpha < x^\beta \Leftrightarrow \exists\, 1 \leq i \leq n : \alpha_1 = \beta_1, \ldots, \alpha_{i-1} = \beta_{i-1}, \alpha_i > \beta_i$.

ds: negative degree reverse lexicographical ordering:
let $\deg(x^\alpha) = \alpha_1 + \cdots + \alpha_n$, then $x^\alpha < x^\beta \Leftrightarrow \deg(x^\alpha) > \deg(x^\beta)$ or
$\deg(x^\alpha) = \deg(x^\beta)$ and $\exists\, 1 \leq i \leq n : \alpha_n = \beta_n, \ldots, \alpha_{i+1} = \beta_{i+1}, \alpha_i > \beta_i$.

Ds: negative degree lexicographical ordering:
let $\deg(x^\alpha) = \alpha_1 + \cdots + \alpha_n$, then $x^\alpha < x^\beta \Leftrightarrow \deg(x^\alpha) > \deg(x^\beta)$ or
$\deg(x^\alpha) = \deg(x^\beta)$ and $\exists\, 1 \leq i \leq n : \alpha_1 = \beta_1, \ldots, \alpha_{i-1} = \beta_{i-1}, \alpha_i < \beta_i$.

ws: (general) weighted reverse lexicographical ordering:
`ws`$(w_1, \ldots, w_n)$, w_1 a nonzero integer, $w_2, \ldots, w_n$ any integer (including 0), is defined as `ds` but with $\deg(x^\alpha) = w_1\alpha_1 + \cdots + w_n\alpha_n$.

Ws: (general) weighted lexicographical ordering:
`Ws`$(w_1, \ldots, w_n)$, w_1 a nonzero integer, $w_2, \ldots, w_n$ any integer (including 0), is defined as `Ds` but with $\deg(x^\alpha) = w_1\alpha_1 + \cdots + w_n\alpha_n$.

B.2.5 Module orderings

SINGULAR offers also orderings on the set of "monomials" $\{x^a e_i \mid a \in N^n, 1 \leq i \leq r\}$ in Loc $K[x]^r$ = Loc $K[x]e_1 + \ldots +$Loc $K[x]e_r$, where $e_1, \ldots, e_r$ denote the canonical generators of Loc $K[x]^r$, the r-fold direct sum of Loc $K[x]$. (The function `gen(i)` yields e_i).

We have two possibilities: either to give priority to the component of a vector in Loc $K[x]^r$ or (which is the default in SINGULAR) to give priority to the coefficients. The orderings `(<,c)` and

`(<,C)` give priority to the coefficients; whereas `(c,<)` and `(C,<)` give priority to the components. Let `<` be any of the monomial orderings of Loc $K[x]$ as above.

(<,C): $<_m = (<, C)$ denotes the module ordering (giving priority to the coefficients):
$x^\alpha e_i <_m x^\beta e_j \Leftrightarrow x^\alpha < x^\beta$ or ($x^\alpha = x^\beta$ and $i < j$).

Example:

```
ring r = 0, (x,y,z), ds;
// the same as ring r = 0, (x,y,z), (ds, C);
[x+y2,z3+xy];
↦ x*gen(1)+xy*gen(2)+y2*gen(1)+z3*gen(2)
[x,x,x];
↦ x*gen(3)+x*gen(2)+x*gen(1)
```

(C,<): $<_m = (C, <)$ denotes the module ordering (giving priority to the component):
$x^\alpha e_i <_m x^\beta e_j \Leftrightarrow i < j$ or ($i = j$ and $x^\alpha < x^\beta$).

Example:

```
ring r = 0, (x,y,z), (C,lp);
[x+y2,z3+xy];
↦ xy*gen(2)+z3*gen(2)+x*gen(1)+y2*gen(1)
[x,x,x];
↦ x*gen(3)+x*gen(2)+x*gen(1)
```

(<,c): $<_m = (<, c)$ denotes the module ordering (giving priority to the coefficients):
$x^\alpha e_i <_m x^\beta e_j \Leftrightarrow x^\alpha < x^\beta$ or ($x^\alpha = x^\beta$ and $i > j$).

Example:

```
ring r = 0, (x,y,z), (lp,c);
[x+y2,z3+xy];
↦ xy*gen(2)+x*gen(1)+y2*gen(1)+z3*gen(2)
[x,x,x];
↦ x*gen(1)+x*gen(2)+x*gen(3)
```

(c,<): $<_m = (c, <)$ denotes the module ordering (giving priority to the component):
$x^\alpha e_i <_m x^\beta e_j \Leftrightarrow i > j$ or ($i = j$ and $x^\alpha < x^\beta$).

Example:

```
ring r = 0, (x,y,z), (c,lp);
[x+y2,z3+xy];
↦ [x+y2,xy+z3]
[x,x,x];
↦ [x,x,x]
```

The output of a vector v in $K[x]^r$ with components $v_1, \ldots, v_r$ has the format $v_1 * gen(1) + \ldots + v_r * gen(r)$ (up to permutation) unless the ordering starts with `c`. In this case a vector is written as $[v_1, \ldots, v_r]$. In all cases Singular can read input in both formats.

B.2.6 Matrix orderings

Let M be an invertible $(n \times n)$-matrix with integer coefficients and $M_1, \ldots, M_n$ the rows of M.

The M-ordering `<` is defined as follows:
$x^a < x^b \Leftrightarrow \exists\ 1 \le i \le n : M_1 a = M_1 b, \ldots, M_{i-1} a = M_{i-1} b$ and $M_i a < M_i b$.

Thus, $x^a < x^b$ if and only if Ma is smaller than Mb with respect to the lexicographical ordering.

The following matrices represent (for 3 variables) the global and local orderings defined above (note that the matrix is not uniquely determined by the ordering):

lp: $\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$ dp: $\begin{pmatrix} 1 & 1 & 1 \\ 0 & 0 & -1 \\ 0 & -1 & 0 \end{pmatrix}$ Dp: $\begin{pmatrix} 1 & 1 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix}$

wp(1,2,3): $\begin{pmatrix} 1 & 2 & 3 \\ 0 & 0 & -1 \\ 0 & -1 & 0 \end{pmatrix}$ Wp(1,2,3): $\begin{pmatrix} 1 & 2 & 3 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix}$

ls: $\begin{pmatrix} -1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & -1 \end{pmatrix}$ ds: $\begin{pmatrix} -1 & -1 & -1 \\ 0 & 0 & -1 \\ 0 & -1 & 0 \end{pmatrix}$ Ds: $\begin{pmatrix} -1 & -1 & -1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix}$

ws(1,2,3): $\begin{pmatrix} -1 & -2 & -3 \\ 0 & 0 & -1 \\ 0 & -1 & 0 \end{pmatrix}$ Ws(1,2,3): $\begin{pmatrix} -1 & -2 & -3 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix}$

Product orderings (see next section) represented by a matrix:

(dp(3), wp(1,2,3)): $\begin{pmatrix} 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & -1 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 2 & 3 \\ 0 & 0 & 0 & 0 & 0 & -1 \\ 0 & 0 & 0 & 0 & -1 & 0 \end{pmatrix}$

(Dp(3), ds(3)): $\begin{pmatrix} 1 & 1 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & -1 & -1 & -1 \\ 0 & 0 & 0 & 0 & 0 & -1 \\ 0 & 0 & 0 & 0 & -1 & 0 \end{pmatrix}$

Orderings with extra weight vector (see below) represented by a matrix:

(dp(3), a(1,2,3),dp(3)): $\begin{pmatrix} 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & -1 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 2 & 3 \\ 0 & 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & -1 \\ 0 & 0 & 0 & 0 & -1 & 0 \end{pmatrix}$

(a(1,2,3,4,5),Dp(3), ds(3)): $\begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 0 \\ 1 & 1 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & -1 & -1 & -1 \\ 0 & 0 & 0 & 0 & 0 & -1 \\ 0 & 0 & 0 & 0 & -1 & 0 \end{pmatrix}$

Example:

```
ring r = 0, (x,y,z), M(1, 0, 0,   0, 1, 0,   0, 0, 1);
```

which may also be written as:

```
intmat m[3][3]=1, 0, 0, 0, 1, 0, 0, 0, 1;
m;
↦ 1,0,0,
↦ 0,1,0,
↦ 0,0,1
```

```
  ring r = 0, (x,y,z), M(m);
  r;
↦ // coefficients: QQ
↦ // number of vars : 3
↦ //        block   1 : ordering M
↦ //                  : names    x y z
↦ //                  : weights  1 0 0
↦ //                  : weights  0 1 0
↦ //                  : weights  0 0 1
↦ //        block   2 : ordering C
```

If the ring has n variables and the matrix does not contain $n \times n$ entries, an error message is given.

WARNING: Singular does not check whether the matrix has full rank. In such a case some computations might not terminate, others may not give a sensible result.

Having these matrix orderings Singular can compute standard bases for any monomial ordering which is compatible with the natural semigroup structure. In practice the global and local orderings together with block orderings should be sufficient in most cases. These orderings are faster than the corresponding matrix orderings, since evaluating a matrix product is time consuming.

B.2.7 Product orderings

Let $x = (x_1, \ldots, x_n)$ and $y = (y_1, \ldots, y_m)$ be two ordered sets of variables, $<_1$ a monomial ordering on $K[x]$ and $<_2$ a monomial ordering on $K[y]$. The product ordering (or block ordering) $< := (<_1, <_2)$ on $K[x, y]$ is the following:
$x^a y^b < x^A y^B \Leftrightarrow x^a <_1 x^A$ or ($x^a = x^A$ and $y^b <_2 y^B$).

Inductively one defines the product ordering of more than two monomial orderings.

In Singular, any of the above global orderings, local orderings or matrix orderings may be combined (in an arbitrary manner and length) to a product ordering. E.g., `(lp(3), M(1, 2, 3, 1, 1, 1, 1, 0, 0), ds(4), ws(1,2,3))` defines: `lp` on the first 3 variables, the matrix ordering `M(1, 2, 3, 1, 1, 1, 1, 0, 0)` on the next 3 variables, `ds` on the next 4 variables and `ws(1,2,3)` on the last 3 variables.

B.2.8 Extra weight vector

`a`$(w_1, \ldots, w_n)$, $w_1, \ldots, w_n$ any integers (including 0), defines $\deg(x^\alpha) = w_1\alpha_1 + \cdots + w_n\alpha_n$ and

$$\deg(x^\alpha) < \deg(x^\beta) \Rightarrow x^\alpha < x^\beta,$$

$$\deg(x^\alpha) > \deg(x^\beta) \Rightarrow x^\alpha > x^\beta.$$

An extra weight vector does not define a monomial ordering by itself: it can only be used in combination with other orderings to insert an extra line of weights into the ordering matrix.

Example:

```
ring r = 0, (x,y,z),  (a(1,2,3),wp(4,5,2));
ring s = 0, (x,y,z),  (a(1,2,3),dp);
ring q = 0, (a,b,c,d),(lp(1),a(1,2,3),ds);
```

B.2.9 Pseudo ordering L

L(max_exponent) is not an ordering but sets the maximal allowed exponent for polynomial in this ring. The default is 32767. The current value for a ring is reflected in the attribute `"maxExp"`. This attribute is also set (and acknowledged) for the list constructed by `ringlist` and the construction of a ring from such a list.

Appendix C Mathematical background

This chapter introduces some of the mathematical notions and definitions used throughout the manual. It is mostly a collection of the most prominent definitions and properties. For details, please, refer to articles or text books (see Section C.9 [References], page 719).

C.1 Standard bases

Definition

Let $R = \mathrm{Loc}_< K[\underline{x}]$ and let I be a submodule of R^r. Note that for r=1 this means that I is an ideal in R. Denote by $L(I)$ the submodule of R^r generated by the leading terms of elements of I, i.e. by $\{L(f) \mid f \in I\}$. Then $f_1, \ldots, f_s \in I$ is called a **standard basis** of I if $L(f_1), \ldots, L(f_s)$ generate $L(I)$.

A standard basis is **minimal** if $\forall i : (f_1, .., f_{i-1}, f_{i+1}, .., f_s) \neq I$.

A minimal standard basis is **completely reduced** if $\forall i :$ `reduce`$(f_i, (f_1, .., f_{i-1}, f_{i+1}, .., f_s)) = f_i$

Properties

normal form:

A function $\mathrm{NF} : R^r \times \{G \mid G \text{ a standard basis}\} \to R^r, (p, G) \mapsto \mathrm{NF}(p|G)$, is called a **normal form** if for any $p \in R^r$ and any standard basis G the following holds: if $\mathrm{NF}(p|G) \neq 0$ then $L(g)$ does not divide $L(\mathrm{NF}(p|G))$ for all $g \in G$. The function may also be applied to any generating set of an ideal: the result is then not uniquely defined.

$\mathrm{NF}(p|G)$ is called a **normal form of p with respect to G**

ideal membership:

For a standard basis G of I the following holds: $f \in I$ if and only if $\mathrm{NF}(f, G) = 0$.

Hilbert function:

Let $I \subseteq K[\underline{x}]^r$ be a homogeneous module, then the Hilbert function H_I of I (see below) and the Hilbert function $H_{L(I)}$ of the leading module $L(I)$ coincide, i.e., $H_I = H_{L(I)}$.

C.2 Hilbert function

Let M $= \bigoplus_{i \in Z} M_i$ be a graded module over $K[x_1, .., x_n]$ with respect to weights $(w_1, ..w_n)$. The **Hilbert function** of M, H_M, is defined (on the integers) by

$$H_M(k) := dim_K M_k.$$

The **Hilbert-Poincare series** of M is the power series

$$\mathrm{HP}_M(t) := \sum_{i=-\infty}^{\infty} H_M(i) t^i = \sum_{i=-\infty}^{\infty} dim_K M_i \cdot t^i.$$

It turns out that $\mathrm{HP}_M(t)$ can be written in two useful ways for weights $(1, .., 1)$:

$$\mathrm{HP}_M(t) = \frac{Q(t)}{(1-t)^n} = \frac{P(t)}{(1-t)^{dim(M)}}$$

where $Q(t)$ and $P(t)$ are polynomials in $\mathbf{Z}[t]$. $Q(t)$ is called the **first Hilbert series**, and $P(t)$ the **second Hilbert series**. If $P(t) = \sum_{k=0}^{N} a_k t^k$, and $d = dim(M)$, then $H_M(s) = \sum_{k=0}^{N} a_k \binom{d+s-k-1}{d-1}$ (the

Hilbert polynomial) for $s \geq N$.

Generalizing this to quasihomogeneous modules we get

$$\mathrm{HP}_M(t) = \frac{Q(t)}{\Pi_{i=1}^n (1 - t^{w_i})}$$

where $Q(t)$ is a polynomial in $\mathbf{Z}[t]$. $Q(t)$ is called the **first (weighted) Hilbert series** of M.

C.3 Syzygies and resolutions

Syzygies

Let R be a quotient of $\mathrm{Loc}_< K[\underline{x}]$ and let $I = (g_1, ..., g_s)$ be a submodule of R^r. Then the **module of syzygies** (or **1st syzygy module**, **module of relations**) of I, $\mathrm{syz}(I)$, is defined to be the kernel of the map $R^s \to R^r$, $\sum_{i=1}^s w_i e_i \mapsto \sum_{i=1}^s w_i g_i$.

The **k-th syzygy module** is defined inductively to be the module of syzygies of the $(k-1)$-st syzygy module.

Note, that the syzygy modules of I depend on a choice of generators $g_1, ..., g_s$. But one can show that they depend on I uniquely up to direct summands.

Example:

```
  ring R= 0,(u,v,x,y,z),dp;
  ideal i=ux, vx, uy, vy;
  print(syz(i));
↦ -y,0, -v,0,
↦ 0, -y,u, 0,
↦ x, 0, 0, -v,
↦ 0, x, 0, u
```

Free resolutions

Let $I = (g_1, ..., g_s) \subseteq R^r$ and $M = R^r/I$. A **free resolution of** M is a long exact sequence

$$... \longrightarrow F_2 \xrightarrow{A_2} F_1 \xrightarrow{A_1} F_0 \longrightarrow M \longrightarrow 0,$$

where the columns of the matrix A_1 generate I . Note that resolutions need not to be finite (i.e., of finite length). The Hilbert Syzygy Theorem states that for $R = \mathrm{Loc}_< K[\underline{x}]$ there exists a ("minimal") resolution of length not exceeding the number of variables.

Example:

```
  ring R= 0,(u,v,x,y,z),dp;
  ideal I = ux, vx, uy, vy;
  resolution resI = mres(I,0); resI;
↦  1      4      4      1
↦ R <--  R <--  R <--  R
↦
↦ 0      1      2      3
↦
  // The matrix A_1 is given by
  print(matrix(resI[1]));
```

```
↦ vy,uy,vx,ux
  // We see that the columns of A_1 generate I.
  // The matrix A_2 is given by
  print(matrix(resI[3]));
↦ u,
↦ -v,
↦ -x,
↦ y
```

Betti numbers and regularity

Let R be a graded ring (e.g., $R = \text{Loc}_< K[\underline{x}]$) and let $I \subset R^r$ be a graded submodule. Let

$$R^r = \bigoplus_a R \cdot e_{a,0} \xleftarrow{A_1} \bigoplus_a R \cdot e_{a,1} \longleftarrow \ldots \longleftarrow \bigoplus_a R \cdot e_{a,n} \longleftarrow 0$$

be a minimal free resolution of R^r/I considered with homogeneous maps of degree 0. Then the **graded Betti number** $b_{i,j}$ of R^r/I is the minimal number of generators $e_{a,j}$ in degree $i+j$ of the j-th syzygy module of R^r/I (i.e., the $(j-1)$-st syzygy module of I). Note that, by definition, the 0-th syzygy module of R^r/I is R^r and the 1st syzygy module of R^r/I is I.

The **regularity** of I is the smallest integer s such that

$$\deg(e_{a,j}) \leq s + j - 1 \quad \text{for all } j.$$

Example:

```
  ring R= 0,(u,v,x,y,z),dp;
  ideal I = ux, vx, uy, vy;
  resolution resI = mres(I,0); resI;
↦  1      4      4      1
↦ R <--  R <--  R <--  R
↦
↦ 0      1      2      3
↦
  // the betti number:
  print(betti(resI), "betti");
↦            0     1     2     3
↦ ------------------------------
↦     0:     1     -     -     -
↦     1:     -     4     4     1
↦ ------------------------------
↦ total:     1     4     4     1
↦
  // the regularity:
  regularity(resI);
↦ 2
```

C.4 Characteristic sets

Let $<$ be the lexicographical ordering on $R = K[x_1, ..., x_n]$ with $x_1 < ... < x_n$. For $f \in R$ let $\text{lvar}(f)$ (the leading variable of f) be the largest variable in f, i.e., if $f = a_s(x_1, ..., x_{k-1})x_k^s + ... + a_0(x_1, ..., x_{k-1})$ for some $k \leq n$ then $\text{lvar}(f) = x_k$.

Moreover, let $\text{ini}(f) := a_s(x_1, ..., x_{k-1})$. The pseudoremainder $r = \text{prem}(g, f)$ of g with respect to f is defined by the equality $\text{ini}(f)^a \cdot g = qf + r$ with $\deg_{lvar(f)}(r) < \deg_{lvar(f)}(f)$ and a minimal.

A set $T = \{f_1, ..., f_r\} \subset R$ is called triangular if $\text{lvar}(f_1) < ... < \text{lvar}(f_r)$. Moreover, let $U \subset T$, then (T, U) is called a triangular system, if T is a triangular set such that $\text{ini}(T)$ does not vanish on $V(T) \setminus V(U) (=: V(T \setminus U))$.

T is called irreducible if for every i there are no d_i, f_i', f_i'' such that

$$\text{lvar}(d_i) < \text{lvar}(f_i) = \text{lvar}(f_i') = \text{lvar}(f_i''),$$

$$0 \notin \text{prem}(\{d_i, \text{ini}(f_i'), \text{ini}(f_i'')\}, \{f_1, ..., f_{i-1}\}),$$

$$\text{prem}(d_i f_i - f_i' f_i'', \{f_1, ..., f_{i-1}\}) = 0.$$

Furthermore, (T, U) is called irreducible if T is irreducible.

The main result on triangular sets is the following: Let $G = \{g_1, ..., g_s\} \subset R$, then there are irreducible triangular sets $T_1, ..., T_l$ such that $V(G) = \bigcup_{i=1}^{l} (V(T_i \setminus I_i))$ where $I_i = \{\text{ini}(f) \mid f \in T_i\}$. Such a set $\{T_1, ..., T_l\}$ is called an **irreducible characteristic series** of the ideal (G).

Example:

```
  ring R= 0,(x,y,z,u),dp;
  ideal i=-3zu+y2-2x+2,
          -3x2u-4yz-6xz+2y2+3xy,
          -3z2u-xu+y2z+y;
  print(char_series(i));
↦ _[1,1],3x2z-y2+2yz,3x2u-3xy-2y2+2yu,
↦ x,     -y+2z,      -2y2+3yu-4
```

C.5 Gauss-Manin connection

Let $f\colon (C^{n+1}, 0) \to (C, 0)$ be a complex isolated hypersurface singularity given by a polynomial with algebraic coefficients which we also denote by f. Let $O = C[x_0, \ldots, x_n]_{(x_0,\ldots,x_n)}$ be the local ring at the origin and J_f the Jacobian ideal of f.

A **Milnor representative** of f defines a differentiable fibre bundle over the punctured disc with fibres of homotopy type of μ n-spheres. The n-th cohomology bundle is a flat vector bundle of dimension n and carries a natural flat connection with covariant derivative ∂_t. The **monodromy operator** is the action of a positively oriented generator of the fundamental group of the punctured disc on the Milnor fibre. Sections in the cohomology bundle of **moderate growth** at 0 form a regular $D = C\{t\}[\partial_t]$-module G, the **Gauss-Manin connection**.

By integrating along flat multivalued families of cycles, one can consider fibrewise global holomorphic differential forms as elements of G. This factors through an inclusion of the **Brieskorn lattice** $H'' := \Omega^{n+1}_{C^{n+1},0} / df \wedge d\Omega^{n-1}_{C^{n+1},0}$ in G.

The D-module structure defines the **V-filtration** V on G by $V^\alpha := \sum_{\beta \geq \alpha} C\{t\} ker(t\partial_t - \beta)^{n+1}$. The Brieskorn lattice defines the **Hodge filtration** F on G by $F_k = \partial_t^k H''$ which comes from the **mixed Hodge structure** on the Milnor fibre. Note that $F_{-1} = H'$.

The induced V-filtration on the Brieskorn lattice determines the **singularity spectrum** Sp by $Sp(\alpha) := \dim_C Gr_V^\alpha Gr_0^F G$. The spectrum consists of μ rational numbers $\alpha_1, \ldots, \alpha_\mu$ such that $e^{2\pi i \alpha_1}, \ldots, e^{2\pi i \alpha_\mu}$ are the eigenvalues of the monodromy. These **spectral numbers** lie in the open interval $(-1, n)$, symmetric about the midpoint $(n-1)/2$.

The spectrum is constant under μ-constant deformations and has the following semicontinuity property: The number of spectral numbers in an interval $(a, a+1]$ of all singularities of a small deformation of f is greater than or equal to that of f in this interval. For semiquasihomogeneous singularities, this also holds for intervals of the form $(a, a+1)$.

Two given isolated singularities f and g determine two spectra and from these spectra we get an integer. This integer is the maximal positive integer k such that the semicontinuity holds for the spectrum of f and k times the spectrum of g. These numbers give bounds for the maximal number of isolated singularities of a specific type on a hypersurface $X \subset P^n$ of degree d: such a hypersurface has a smooth hyperplane section, and the complement is a small deformation of a cone over this hyperplane section. The cone itself being a μ-constant deformation of $x_0^d + \ldots + x_n^d = 0$, the singularities are bounded by the spectrum of $x_0^d + \ldots + x_n^d$.

Using the library `gmssing.lib` one can compute the **monodromy**, the V-fitration on H''/H', and the spectrum.

Let us consider as an example $f = x^5 + x^2y^2 + y^5$. First, we compute a matrix M such that $\exp(2\pi i M)$ is a monodromy matrix of f and the Jordan normal form of M :

```
  LIB "mondromy.lib";
  ring R=0,(x,y),ds;
  poly f=x5+x2y2+y5;
  matrix M=monodromyB(f);
  print(M);
↦ 11/10,0,    0,    0,    0,   0,-1/4,0,   0,   0,  0,
↦ 0,    13/10,0,    0,    0,   0,0,   15/8,0,   0,  0,
↦ 0,    0,    13/10,0,    0,   0,0,   0,   15/8,0,  0,
↦ 0,    0,    0,    11/10,-1/4,0,0,   0,   0,   0,  0,
↦ 0,    0,    0,    0,    9/10,0,0,   0,   0,   0,  0,
↦ 0,    0,    0,    0,    0,   1,0,   0,   0,   0,  3/5,
↦ 0,    0,    0,    0,    0,   0,9/10,0,   0,   0,  0,
↦ 0,    0,    0,    0,    0,   0,0,   7/10,0,   0,  0,
↦ 0,    0,    0,    0,    0,   0,0,   0,   7/10,0,  0,
↦ 0,    0,    0,    0,    0,   0,0,   0,   0,   1,  -2/5,
↦ 0,    0,    0,    0,    0,   0,0,   0,   0,   5/8,0
```

Now, we compute the V-fitration on H''/H' and the spectrum:

```
  LIB "gmssing.lib";
  ring R=0,(x,y),ds;
  poly f=x5+x2y2+y5;
  list l=vfilt(f);
  print(l[1]); // spectral numbers
↦ -1/2,
↦ -3/10,
↦ -1/10,
↦ 0,
↦ 1/10,
↦ 3/10,
↦ 1/2
  print(l[2]); // corresponding multiplicities
↦ 1,
↦ 2,
↦ 2,
↦ 1,
↦ 2,
↦ 2,
↦ 1
  print(l[3]); // vector space of i-th graded part
↦ [1]:
↦    _[1]=gen(11)
```

```
↦ [2]:
↦    _[1]=gen(10)
↦    _[2]=gen(6)
↦ [3]:
↦    _[1]=gen(9)
↦    _[2]=gen(4)
↦ [4]:
↦    _[1]=gen(5)
↦ [5]:
↦    _[1]=gen(3)
↦    _[2]=gen(8)
↦ [6]:
↦    _[1]=gen(2)
↦    _[2]=gen(7)
↦ [7]:
↦    _[1]=gen(1)
  print(l[4]); // monomial vector space basis of H''/s*H''
↦ y5,
↦ y4,
↦ y3,
↦ y2,
↦ xy,
↦ y,
↦ x4,
↦ x3,
↦ x2,
↦ x,
↦ 1
  print(l[5]); // standard basis of Jacobian ideal
↦ 2x2y+5y4,
↦ 5x5-5y5,
↦ 2xy2+5x4,
↦ 10y6+25x3y4
```

Here `l[1]` contains the spectral numbers, `l[2]` the corresponding multiplicities, `l[3]` a C -basis of the V-filtration on H''/H' in terms of the monomial basis of $O/J_f \cong H''/H'$ in `l[4]` (seperated by degree).

If the principal part of f is C-nondegenerate, one can compute the spectrum using the library `spectrum.lib`. In this case, the V-filtration on H'' coincides with the Newton-filtration on H'' which allows to compute the spectrum more efficiently.

Let us calculate one specific example, the maximal number of triple points of type $\tilde{E}_6$ on a surface $X \subset P^3$ of degree seven. This calculation can be done over the rationals. We choose a local ordering on $Q[x,y,z]$. Here we take the negative degree lexicographical ordering, in Singular denoted by `ds`:

```
ring r=0,(x,y,z),ds;
LIB "spectrum.lib";
poly f=x^7+y^7+z^7;
list s1=spectrumnd( f );
s1;
↦ [1]:
↦    _[1]=-4/7
↦    _[2]=-3/7
↦    _[3]=-2/7
```

```
↦     _[4]=-1/7
↦     _[5]=0
↦     _[6]=1/7
↦     _[7]=2/7
↦     _[8]=3/7
↦     _[9]=4/7
↦     _[10]=5/7
↦     _[11]=6/7
↦     _[12]=1
↦     _[13]=8/7
↦     _[14]=9/7
↦     _[15]=10/7
↦     _[16]=11/7
↦  [2]:
↦     1,3,6,10,15,21,25,27,27,25,21,15,10,6,3,1
```

The command `spectrumnd(f)` computes the spectrum of f and returns a list with six entries: The Milnor number $\mu(f)$, the geometric genus $p_g(f)$ and the number of different spectrum numbers. The other three entries are of type `intvec`. They contain the numerators, denominators and multiplicities of the spectrum numbers. So $x^7+y^7+z^7=0$ has Milnor number 216 and geometrical genus 35. Its spectrum consists of the 16 different rationals
$\frac{3}{7}, \frac{4}{7}, \frac{5}{7}, \frac{6}{7}, \frac{1}{1}, \frac{8}{7}, \frac{9}{7}, \frac{10}{7}, \frac{11}{7}, \frac{12}{7}, \frac{13}{7}, \frac{2}{1}, \frac{15}{7}, \frac{16}{7}, \frac{17}{7}, \frac{18}{7}$
appearing with multiplicities
1,3,6,10,15,21,25,27,27,25,21,15,10,6,3,1.

The singularities of type $\tilde{E}_6$ form a μ-constant one parameter family given by $x^3+y^3+z^3+\lambda xyz = 0, \quad \lambda^3 \neq -27$. Therefore they have all the same spectrum, which we compute for $x^3+y^3+z^3$.

```
poly g=x^3+y^3+z^3;
list s2=spectrumnd(g);
s2;
↦  [1]:
↦     8
↦  [2]:
↦     1
↦  [3]:
↦     4
↦  [4]:
↦     1,4,5,2
↦  [5]:
↦     1,3,3,1
↦  [6]:
↦     1,3,3,1
```

Evaluating semicontinuity is very easy:

```
semicont(s1,s2);
↦  18
```

This tells us that there are at most 18 singularities of type $\tilde{E}_6$ on a septic in P^3. But $x^7+y^7+z^7$ is semiquasihomogeneous (sqh), so we can also apply the stronger form of semicontinuity:

```
semicontsqh(s1,s2);
↦  17
```

So in fact a septic has at most 17 triple points of type $\tilde{E}_6$.

Note that `spectrumnd(f)` works only if f has a nondegenerate principal part. In fact `spectrumnd` will detect a degenerate principal part in many cases and print out an error message. However if it is known in advance that f has nondegenerate principal part, then the spectrum may be computed much faster using `spectrumnd(f,1)`.

C.6 Toric ideals and integer programming

C.6.1 Toric ideals

Let A denote an $m \times n$ matrix with integral coefficients. For $u \in \mathbb{Z}^n$, we define u^+, u^- to be the uniquely determined vectors with nonnegative coefficients and disjoint support (i.e., $u_i^+ = 0$ or $u_i^- = 0$ for each component i) such that $u = u^+ - u^-$. For $u \geq 0$ component-wise, let x^u denote the monomial $x_1^{u_1} \cdot \ldots \cdot x_n^{u_n} \in K[x_1, \ldots, x_n]$.

The ideal

$$I_A := (x^{u^+} - x^{u^-} | u \in \ker(A) \cap \mathbb{Z}^n) \ \subset K[x_1, \ldots, x_n]$$

is called a **toric ideal.**

The first problem in computing toric ideals is to find a finite generating set: Let $v_1, \ldots, v_r$ be a lattice basis of $\ker(A) \cap \mathbb{Z}^n$ (i.e, a basis of the $\mathbb{Z}$-module). Then

$$I_A := I : (x_1 \cdot \ldots \cdot x_n)^\infty$$

where

$$I =< x^{v_i^+} - x^{v_i^-} | i = 1, \ldots, r >$$

The required lattice basis can be computed using the LLL-algorithm (Section 5.1.151 [system], page 270, see see [[Coh93]], page 712). For the computation of the saturation, there are various possibilities described in the section Algorithms.

C.6.2 Algorithms

The following algorithms are implemented in Section D.4.29 [toric_lib], page 762.

C.6.2.1 The algorithm of Conti and Traverso

The algorithm of Conti and Traverso (see [[CoTr91]], page 712) computes I_A via the extended matrix $B = (I_m|A)$, where I_m is the $m \times m$ unity matrix. A lattice basis of B is given by the set of vectors $(a^j, -e_j) \in \mathbb{Z}^{m+n}$, where a^j is the j-th row of A and e_j the j-th coordinate vector. We look at the ideal in $K[y_1, \ldots, y_m, x_1, \ldots, x_n]$ corresponding to these vectors, namely

$$I_1 =< y^{a_j^+} - x_j y^{a_j^-} | j = 1, \ldots, n > .$$

We introduce a further variable t and adjoin the binomial $t \cdot y_1 \cdot \ldots \cdot y_m - 1$ to the generating set of I_1, obtaining an ideal I_2 in the polynomial ring $K[t, y_1, \ldots, y_m, x_1, \ldots, x_n]$. I_2 is saturated w.r.t. all variables because all variables are invertible modulo I_2. Now I_A can be computed from I_2 by eliminating the variables $t, y_1, \ldots, y_m$.

Because of the big number of auxiliary variables needed to compute a toric ideal, this algorithm is rather slow in practice. However, it has a special importance in the application to integer programming (see Section C.6.4 [Integer programming], page 711).

C.6.2.2 The algorithm of Pottier

The algorithm of Pottier (see [[Pot94]], page 712) starts by computing a lattice basis $v_1, \dots, v_r$ for the integer kernel of A using the LLL-algorithm (Section 5.1.151 [system], page 270). The ideal corresponding to the lattice basis vectors

$$I_1 =< x^{v_i^+} - x^{v_i^-} | i = 1, \dots, r >$$

is saturated – as in the algorithm of Conti and Traverso – by inversion of all variables: One adds an auxiliary variable t and the generator $t \cdot x_1 \cdot \ldots \cdot x_n - 1$ to obtain an ideal I_2 in $K[t, x_1, \dots, x_n]$ from which one computes I_A by elimination of t.

C.6.2.3 The algorithm of Hosten and Sturmfels

The algorithm of Hosten and Sturmfels (see [[HoSt95]], page 712) allows to compute I_A without any auxiliary variables, provided that A contains a vector w with positive coefficients in its row space. This is a real restriction, i.e., the algorithm will not necessarily work in the general case.

A lattice basis $v_1, \dots, v_r$ is again computed via the LLL-algorithm. The saturation step is performed in the following way: First note that w induces a positive grading w.r.t. which the ideal

$$I =< x^{v_i^+} - x^{v_i^-} | i = 1, \dots, r >$$

is homogeneous corresponding to our lattice basis. We use the following lemma:

Let I be a homogeneous ideal w.r.t. the weighted reverse lexicographical ordering with weight vector w and variable order $x_1 > x_2 > \ldots > x_n$. Let G denote a Groebner basis of I w.r.t. this ordering. Then a Groebner basis of $(I : x_n^\infty)$ is obtained by dividing each element of G by the highest possible power of x_n.

From this fact, we can succesively compute

$$I_A = I : (x_1 \cdot \ldots \cdot x_n)^\infty = (((I : x_1^\infty) : x_2^\infty) : \ldots : x_n^\infty);$$

in the i-th step we take x_i as the smallest variable and apply the lemma with x_i instead of x_n.

This procedure involves n Groebner basis computations. Actually, this number can be reduced to at most $n/2$ (see [[HoSh98]], page 712), and each computation – except for the first one – proves to be simple and fast in practice.

C.6.2.4 The algorithm of Di Biase and Urbanke

Like the algorithm of Hosten and Sturmfels, the algorithm of Di Biase and Urbanke (see [[DBUr95]], page 712) performs up to $n/2$ Groebner basis computations. It needs no auxiliary variables, but a supplementary precondition; namely, the existence of a vector without zero components in the kernel of A.

The main idea comes from the following observation:

Let B be an integer matrix, $u_1, \dots, u_r$ a lattice basis of the integer kernel of B. Assume that all components of u_1 are positive. Then

$$I_B =< x^{u_i^+} - x^{u_i^-} | i = 1, \dots, r >,$$

i.e., the ideal on the right is already saturated w.r.t. all variables.

The algorithm starts by finding a lattice basis $v_1, \dots, v_r$ of the kernel of A such that v_1 has no zero component. Let $\{i_1, \dots, i_l\}$ be the set of indices i with $v_{1,i} < 0$. Multiplying the components

$i_1, \ldots, i_l$ of $v_1, \ldots, v_r$ and the columns $i_1, \ldots, i_l$ of A by -1 yields a matrix B and a lattice basis $u_1, \ldots, u_r$ of the kernel of B that fulfill the assumption of the observation above. It is then possible to compute a generating set of I_A by applying the following "variable flip" successively to $i = i_1, \ldots, i_l$:

Let $>$ be an elimination ordering for x_i. Let A_i be the matrix obtained by multiplying the i-th column of A by -1. Let

$$\{x_i^{r_j} x^{a_j} - x^{b_j} | j \in J\}$$

be a Groebner basis of I_{A_i} w.r.t. $>$ (where x_i is neither involved in x^{a_j} nor in x^{b_j}). Then

$$\{x^{a_j} - x_i^{r_j} x^{b_j} | j \in J\}$$

is a generating set for I_A.

C.6.2.5 The algorithm of Bigatti, La Scala and Robbiano

The algorithm of Bigatti, La Scala and Robbiano (see [[BLR98]], page 712) combines the ideas of the algorithms of Pottier and of Hosten and Sturmfels. The computations are performed on a graded ideal with one auxiliary variable u and one supplementary generator $x_1 \cdot \ldots \cdot x_n - u$ (instead of the generator $t \cdot x_1 \cdot \ldots \cdot x_n - 1$ in the algorithm of Pottier). The algorithm uses a quite unusual technique to get rid of the variable u again.

There is another algorithm of the authors which tries to parallelize the computations (but which is not implemented in this library).

C.6.3 The Buchberger algorithm for toric ideals

Toric ideals have a very special structure that allows us to improve the Buchberger algorithm in many aspects: They are prime ideals and generated by binomials. Pottier used this fact to describe all operations of the Buchberger algorithm on the ideal generators in terms of vector additions and subtractions. Some other strategies like multiple reduction (see [[CoTr91]], page 712) or the use of bit vectors to represent the support of a monomial (see [[Big97]], page 712) may be applied to more general ideals, but show to be especially useful in the toric case.

C.6.4 Integer programming

Let A be an $m \times n$ matrix with integral coefficients, $b \in \mathbb{Z}^m$ and $c \in \mathbb{Z}^n$. The problem

$$\min\{c^T x | x \in \mathbb{Z}^n, Ax = b, x \geq 0 \text{ component-wise}\}$$

is called an instance of the **integer programming problem** or **IP problem.**

The IP problem is very hard; namely, it is NP-complete.

For the following discussion let $c \geq 0$ (component-wise). We consider c as a weight vector; because of its nonnegativity, c can be refined into a monomial ordering $>_c$. It turns out that we can solve such an IP instance with the help of toric ideals:

First we assume that an initial solution v (i.e., $v \in \mathbb{Z}^n, v \geq 0, Av = b$) is already known. We obtain the optimal solution v_0 (i.e., with $c^T v_0$ minimal) by the following procedure:

- (1) Compute the toric ideal I(A) using one of the algorithms in the previous section.
- (2) Compute the reduced Groebner basis G(c) of I(A) w.r.t. $>_c$.
- (3) Reduce x^v modulo G(c) using the Hironaka division algorithm. If the result of this reduction is x^w , then w is an optimal solution of the given instance.

If no initial solution is known, we are nevertheless able to solve the problem with similar techniques. For this purpose we replace our instance by an extended instance with the matrix used in the Conti-Traverso algorithm. Indeed, the Conti-Traverso algorithm offers the possibility to verify solvability of a given instance and to find an initial solution in the case of existence (but none of the other algorithms does!). Details can be found in see [[CoTr91]], page 712 and see [[The99]], page 712.

An implementation of the above algorithm and some examples can be found in Section D.4.10 [intprog_lib], page 747.

In general, classical methods for solving IP instances like Branch-and-Bound methods seem to be faster than the methods using toric ideals. But the latter have one great advantage: If one wants to solve various instances that differ only by the vector b , one has to perform steps (1) and (2) above only once. As the running time of step (3) is very short, solving all the instances is not much harder than solving one single instance.

For a detailed discussion see see [[The99]], page 712.

C.6.5 Relevant References

- [Big97] Bigatti, A.M.: Computation of Hilbert-Poincare series. Journal of Pure and Applied Algebra (1997) 199, 237-253
- [BLR98] Bigatti, A.M.; La Scala, R.; Robbiano, L.: Computing toric ideals. Journal of Symbolic Computation (1999) 27, 351-366
- [Coh93] Cohen, H.: A Course in Computational Algebraic Number Theory. Springer (1997)
- [CoTr91] Conti, P.; Traverso, C.: Buchberger algorithm and integer programming. Proceedings AAECC-9 (new Orleans), Springer LNCS (1991) 539, 130-139
- [DBUr95] Di Biase, F.; Urbanke, R.: An algorithm to calculate the kernel of certain polynomial ring homomorphisms. Experimental Mathematics (1995) 4, 227-234
- [HoSh98] Hosten, S.; Shapiro, J.: Primary decomposition of lattice basis ideals. Journal of Symbolic Computation (2000), 29, 625-639
- [HoSt95] Hosten, S.; Sturmfels, B.: GRIN: An implementation of Groebner bases for integer programming. in Balas, E.; Clausen, J. (editors): Integer Programming and Combinatorial Optimization. Springer LNCS (1995) 920, 267-276
- [Pot94] Pottier, L.: Groebner bases of toric ideals. Rapport de recherche 2224 (1997), INRIA Sophia Antipolis
- [Stu96] Sturmfels, B.: Groebner Bases and Convex Polytopes. University Lecture Series, Volume 8 (1996), American Mathematical Society
- [The99] Theis, C.: Der Buchberger-Algorithmus fuer torische Ideale und seine Anwendung in der ganzzahligen Optimierung. Diplomarbeit, Universitaet des Saarlandes (1999), Saarbruecken (Germany)

C.7 Non-commutative algebra

See Section 7.4 [Mathematical background (plural)], page 359.

C.8 Decoding codes with Groebner bases

This section introduces some of the mathematical notions, definitions, and results for solving decoding problems and finding the minimum distance of linear (and in particular cyclic) codes. The material presented here should assist the user in working with Section D.10.2 [decodegb_lib], page 797. More details can be obtained from [[BP2008b]], page 718.

C.8.1 Codes and the decoding problem

Codes

- Let F_q be a field with q elements. A *linear code* C is a linear subspace of F_q^n endowed with the **Hamming metric**.
- **Hamming distance** between **x**,**y**$\in F_q^n : d(x,y) = \#\{i | x_i \neq y_i\}$. **Hamming weight** of **x**$\in F_q^n$: $wt(x) = \#\{i | x_i \neq 0\}$.
- **Minimum distance** of the code $C : d(C) := \min_{\mathbf{x},\mathbf{y}\in C, \mathbf{x}\neq\mathbf{y}}(d(\mathbf{x},\mathbf{y}))$.
- The code C of dimension k and minimum distance d is denoted as $[n,k,d]$.
- A matrix G whose rows are the base vectors of C is the **generator matrix**.
- A matrix H with the property $\mathbf{c} \in C \iff H\mathbf{c}^T = 0$ is the **check matrix**.

Cyclic codes

The code C is **cyclic**, if for every codeword $\mathbf{c} = (c_0, \ldots, c_{n-1})$ in C its cyclic shift $(c_{n-1}, c_0, \ldots, c_{n-2})$ is again a codeword in C. When working with cyclic codes, vectors are usually presented as polynomials. So $\mathbf{c}$ is represented by the polynomial $c(x) = \sum_{i=0}^{n-1} c_i x^i$ with $x^n = 1$, more precisely $c(x)$ is an element of the factor ring $F_q[X]/\langle X^n - 1\rangle$. Cyclic codes over F_q of length n correspond one-to-one to ideals in this factor ring. We assume for cyclic codes that $(q,n) = 1$. Let $F = F_{q^m}$ be the splitting field of $X^n - 1$ over F_q. Then F has a **primitive** n**-th root of unity** which will be denoted by a. A cyclic code is uniquely given by a **defining set** S_C which is a subset of $\mathbb{Z}_n$ such that

$$c(x) \in C \text{ if } c(a^i) = 0 \text{ for all } i \in S_C.$$

A cyclic code has several defining sets.

Decoding problem

- **Complete decoding**: Given $y \in F_q^n$ and a code $C \subseteq F_q^n$, so that y is at distance $d(y,C)$ from the code, find $c \in C : d(y,c) = d(y,C)$.
- **Bounded up to half the minimum distance**: With the additional assumption $d(\mathbf{y},C) \leq (d(C)-1)/2$, a codeword with the above property is unique.

Decoding via systems solving

One distinguishes between two concepts:

- **Generic decoding**: Solve some system $S(C)$ and obtain some "closed" formulas CF. Evaluating these formulas at data specific to a received word $\mathbf{r}$ should yield a solution to the decoding problem. For example for $f \in CF : f(syndrome(\mathbf{r}), x) = poly(x)$. The roots of $poly(x) = 0$ yield error positions, see the section on the general error-locator polynomial.
- **Online decoding**: Solve some system $S(C, \mathbf{r})$. The solutions should solve the decoding problem.

Computational effort

- Generic decoding. Here, preprocessing is very hard, whereas decoding is relatively simple (if the formulas are sparse).
- Online decoding. In this case, decoding is the hard part.

C.8.2 Cooper philosophy

Computing syndromes in cyclic code case

Let C be an $[n,k]$ cyclic code over F_q; F is a splitting field with a being a primitive n-th root of unity. Let $S_C = \{i_1,\ldots,i_{n-k}\}$ be the complete defining set of C. Let $\mathbf{r} = \mathbf{c} + \mathbf{e}$ be a received word with $\mathbf{c} \in C$ and $\mathbf{e}$ an error vector. Denote the corresponding polynomials in $F_q[x]/\langle x^n - 1\rangle$ by $r(x)$, $c(x)$ and $e(x)$, resp. Compute syndromes

$$s_{i_m} = r(a^{i_m}) = e(a^{i_m}) = \sum_{l=1}^{t} e_{j_l}(a^{i_m})^{j_l}, \quad 1 \le m \le n-k,$$

where t is the number of errors, $j_1,\ldots,j_t$ are the **error positions** and $e_{j_1},\ldots,e_{j_t}$ are the **error values**. Define $z_l = a^{j_l}$ and $y_l = e_{j_l}$. Then $z_1,\ldots,z_t$ are the error locations and $y_1,\ldots,y_t$ are the error values and the syndromes above become **generalized power sum functions** $s_{i_m} = \sum_{l=1}^{t} y_l z_l^{i_m}$, $1 \le m \le n-k$.

CRHT-ideal

Replace the concrete values above by variables and add some natural restrictions. Introduce

- $f_u := \sum_{l=1}^{e} Y_l Z_l^{i_u} - X_u = 0, 1 \le u \le n-k$;
- $\epsilon_j := X_j^{q^m} - X_j = 0, 1 \le j \le n-k$, since $s_j \in F$;
- $\eta_i := Z_i^{n+1} - Z_i = 0, 1 \le i \le e$, since a^{j_l} are either n-th roots of unity or zero;
- $\lambda_i := Y_i^{q-1} - 1 = 0, 1 \le i \le e$, since $y_l \in F_q \setminus \{0\}$.

We obtain the following set of polynomials in the variables $X = (X_1,\ldots,X_{n-k})$, $Z = (Z_1,\ldots,Z_e)$ and $Y = (Y_1,\ldots,Y_e)$:

$$F_C = \{f_j, \epsilon_j, \eta_i, \lambda_i : 1 \le j \le n-k, 1 \le i \le e\} \subset F_q[X,Z,Y].$$

The zero-dimensional ideal I_C generated by F_C is the **CRHT-syndrome ideal** associated to the code C, and the variety $V(F_C)$ defined by F_C is the **CRHT-syndrome variety**, after Chen, Reed, Helleseth and Truong.

General error-locator polynomial

Adding some more polynomials to F_C, thus obtaining some F'_C, it is possible to prove the following **Theorem**:

Every cyclic code C possesses a **general error-locator polynomial** L_C from $F_q[X_1,\ldots,X_{n-k},Z]$ that satisfies the following two properties:

- $L_C = Z^e + a_{t-1}Z^{e-1} + \ldots + a_0$ with $a_j \in F_q[X_1,\ldots,X_{n-k}]$, $0 \le j \le e-1$, where e is the error-correcting capacity;
- given a syndrome $\mathbf{s} = (s_{i_1},\ldots,s_{i_{n-k}}) \in F^{n-k}$ corresponding to an error of weight $t \le e$ and error locations $\{k_1,\ldots,k_t\}$, if we evaluate the $X_u = s_{i_u}$ for all $1 \le u \le n-k$, then the roots of $L_C(\mathbf{s},Z)$ are exactly $a^{k_1},\ldots,a^{k_t}$ and 0 of multiplicity $e-t$, in other words $L_C(\mathbf{s},Z) = Z^{e-t}\prod_{i=1}^{t}(Z - a^{k_i})$.

The general error-locator polynomial actually is an element of the reduced Gröbner basis of $\langle F'_C\rangle$. Having this polynomial, decoding of the cyclic code C reduces to univariate factorization.

For an example see `sysCRHT` in Section D.10.2 [decodegb_lib], page 797. More on Cooper's philosophy and the general error-locator polynomial can be found in [[OS2005]], page 718.

Finding the minimum distance

The method described above can be adapted to find the minimum distance of a code. More concretely, the following holds:

Let C be the binary $[n,k,d]$ cyclic code with the defining set $S_C = \{i_1, \ldots, i_v\}$. Let $1 \le w \le n$ and let $J_C(w)$ denote the system:

$$\begin{aligned} Z_1^{i_1} + \ldots + Z_w^{i_1} &= 0, \\ &\vdots \\ Z_1^{i_v} + \ldots + Z_w^{i_v} &= 0, \\ Z_1^n - 1 &= 0, \\ &\vdots \\ Z_w^n - 1 &= 0, \\ p(n, Z_i, Z_j) &= 0, 1 \le i < j \le w. \end{aligned}$$

Then the number of solutions of $J_C(w)$ is equal to $w!$ times the number of codewords of weight w. And for $1 \le w \le d$, either $J_C(w)$ has no solutions, which is equivalent to $w < d$, or $J_C(w)$ has some solutions, which is equivalent to $w = d$.

For an example see `sysCRHTMindist` in Section D.10.2 [decodegb_lib], page 797. More on finding the minimum distance with Groebner bases can be found in [[S2007]], page 718. See [[OS2005]], page 718, for the definition of the polynomial p above.

C.8.3 Generalized Newton identities

The **error-locator polynomial** is defined by

$$\sigma(Z) = \prod_{l=1}^{t} (Z - z_l).$$

If this product is expanded,

$$\sigma(Z) = Z^t + \sigma_1 Z^{t-1} + \ldots + \sigma_{t-1} Z + \sigma_t,$$

then the coefficients σ_i are the **elementary symmetric functions** in the error locations $z_1, \ldots, z_t$

$$\sigma_i = (-1)^i \sum_{1 \le j_1 < j_2 < \ldots < j_i \le t} z_{j_1} z_{j_2} \ldots z_{j_i}, \ 1 \le i \le t.$$

Generalized Newton identities

The syndromes $s_i = r(a^i) = e(a^i)$ and the coefficients σ_i satisfy the following **generalized Newton identities**:

$$s_i + \sum_{j=1}^{t} \sigma_j s_{i-j} = 0, \quad \text{for all} \quad i \in \mathbb{Z}_n.$$

Decoding up to error-correcting capacity

We have $s_{i+n} = s_i$, for all $i \in \mathbb{Z}_n$, since $s_{i+n} = r(a^{i+n}) = r(a^i)$. Furthermore

$$s_i^q = (e(a^i))^q = e(a^{iq}) = s_{qi}, \text{ for all } i \in Z_n,$$

and $\sigma_i^{q^m} = \sigma_i$, for all $1 \leq i \leq t$. Replace the syndromes by variables and obtain the following set of polynomials $Newton_t$ in the variables $S_1, \ldots, S_n$ and $\sigma_1, \ldots, \sigma_t$:

$$\sigma_i^{q^m} - \sigma_i, \quad \forall\, 1 \leq i \leq t,$$

$$S_{i+n} - S_i, \quad \forall\, i \in \mathbb{Z}_n,$$

$$S_i^q - S_{qi}, \quad \forall\, i \in \mathbb{Z}_n,$$

$$S_i + \sum_{j=1}^{t} \sigma_j S_{i-j}, \quad \forall\, i \in \mathbb{Z}_n,$$

$$S_i - s_i(r) \quad \forall\, i \in S_C.$$

For an example see `sysNewton` in Section D.10.2 [decodegb_lib], page 797. More on this method and the method based on Waring function can be found in [[ABF2002]], page 718. See also [[ABF2008]], page 718.

C.8.4 Fitzgerald-Lax method

Affine codes

Let $I = \langle g_1, \ldots, g_m \rangle \subseteq F_q[X_1, \ldots, X_s]$ be an ideal. Define

$$I_q := I + \langle X_1^q - X_1, \ldots, X_s^q - X_s \rangle.$$

So I_q is a zero-dimensional ideal. Define also $V(I_q) =: \{P_1, \ldots, P_n\}$. Every q-ary linear code C with parameters $[n, k]$ can be seen as an **affine variety code** $C(I, L)$, that is, the image of a vector space L of the **evaluation map**

$$\phi : R \to F_q^n$$

$$\bar{f} \mapsto (f(P_1), \ldots, f(P_n)),$$

where $R := F_q[U_1, \ldots, U_s]/I_q$, L is a vector subspace of R and $\bar{f}$ the coset of f in $F_q[U_1, \ldots, U_s]$ modulo I_q.

Decoding affine variety codes

Given a q-ary $[n, k]$ code C with a generator matrix $G = (g_{ij})$:

1. choose s, such that $q^s \geq n$, and construct s distinct points $P_1, \ldots, P_s$ in F_q^s.
2. Construct a Gröbner basis $\{g_1, \ldots, g_m\}$ for an ideal I of polynomials from $F_q[X_1, \ldots, X_s]$ that vanish at the points $P_1, \ldots, P_s$. Define $\xi_i \in F_q[X_1, \ldots, X_s]$ such that $\xi_i(P_i) = 1, \xi_i(P_j) = 0, i \neq j$.
3. Then $f_i = \sum_{i=1}^{n} g_{ij} \xi_j$ span the space L, so that $g_{ij} = f_i(P_j)$.

In this way we obtain that the code C is the image of the evaluation above, thus $C = C(I, L)$. In the same way by considering a parity check matrix instead of a generator matrix we have that the dual code is also an affine variety code.

The method of decoding is a generalization of CRHT. One needs to add polynomials $(g_l(X_{k1}, \ldots, X_{ks}))_{l=1,\ldots,m;k=1,\ldots,t}$ for every error position. We also assume that field equations on X_{ij}'s are included among the polynomials above. Let C be a q-ary $[n, k]$ linear code such that its dual is written as an affine variety code of the form $C^\perp = C(I, L)$. Let $\mathbf{r} = \mathbf{c}+\mathbf{e}$ as usual and $t \leq e$. Then the syndromes are computed by $s_i = \sum_{j=1}^n r_j f_i(P_j) = \sum_{j=1}^n e_j f_i(P_j)$ for $i = 1, \ldots, n-k$.

Consider the ring $F_q[X_{11}, \ldots, X_{1s}, \ldots, X_{t1}, \ldots, X_{ts}, E_1, \ldots, E_t]$, where $(X_{i1}, \ldots, X_{is})$ correspond to the i-th error position and E_i to the i-th error value. Consider the ideal Id_C generated by

$$\sum_{j=1}^t E_j f_i(X_{j1}, \ldots, X_{js}) - s_i, 1 \leq i \leq n-k,$$

$$g_l(X_{j1}, \ldots, X_{js}), 1 \leq l \leq m,$$

$$E_k^{q-1} - 1.$$

Theorem: Let G be the reduced Gröbner basis for Id_C with respect to an elimination order $X_{11} < \ldots < X_{1s} < E_1$. Then we may solve for the error locations and values by applying elimination theory to the polynomials in G.

For an example see `sysFL` in Section D.10.2 [decodegb_lib], page 797. More on this method can be found in [[FL1998]], page 718.

C.8.5 Decoding method based on quadratic equations

Preliminary definitions

Let $\mathbf{b}_1, \ldots, \mathbf{b}_n$ be a basis of F_q^n and let B be the $n \times n$ matrix with $\mathbf{b}_1, \ldots, \mathbf{b}_n$ as rows. The **unknown syndrome** $\mathbf{u}(B, \mathbf{e})$ of a word $\mathbf{e}$ w.r.t B is the column vector $\mathbf{u}(B, \mathbf{e}) = B\mathbf{e}^T$ with entries $u_i(B, \mathbf{e}) = \mathbf{b}_i \cdot \mathbf{e}$ for $i = 1, \ldots, n$.

For two vectors $\mathbf{x}, \mathbf{y} \in F_q^n$ define $\mathbf{x} * \mathbf{y} = (x_1y_1, \ldots, x_ny_n)$. Then $\mathbf{b}_i * \mathbf{b_j}$ is a linear combination of $\mathbf{b}_1, \ldots, \mathbf{b}_n$, so there are constants $\mu_l^{ij} \in F_q$ such that $\mathbf{b}_i * \mathbf{b}_j = \sum_{l=1}^n \mu_l^{ij} \mathbf{b}_l$. The elements $\mu_l^{ij} \in F_q$ are the **structure constants** of the basis $\mathbf{b}_1, \ldots, \mathbf{b}_n$.

Let B_s be the $s \times n$ matrix with $\mathbf{b}_1, \ldots, \mathbf{b}_s$ as rows ($B = B_n$). Then $\mathbf{b}_1, \ldots, \mathbf{b}_n$ is an **ordered MDS basis** and B an **MDS matrix** if all the $s \times s$ submatrices of B_s have rank s for all $s = 1, \ldots, n$.

Expressing known syndromes

Let C be an F_q-linear code with parameters $[n, k, d]$. W.l.o.g $n \leq q$. H is a check matrix of C. Let $\mathbf{h}_1, \ldots, \mathbf{h}_{n-k}$ be the rows of H. One can express $\mathbf{h}_i = \sum_{j=1}^n a_{ij}\mathbf{b}_j$ with some $a_{ij} \in F_q$. In other words $H = AB$ where A is the $(n-k) \times n$ matrix with entries a_{ij}.

Let $\mathbf{r} = \mathbf{c}+\mathbf{e}$ be a received word with $\mathbf{c} \in C$ and $\mathbf{e}$ an error vector. The syndromes of $\mathbf{r}$ and $\mathbf{e}$ w.r.t H are equal and known:

$$s_i(\mathbf{r}) := \mathbf{h}_i \cdot \mathbf{r} = \mathbf{h}_i \cdot \mathbf{e} = s_i(\mathbf{e}).$$

They can be expressed in the unknown syndromes of $\mathbf{e}$ w.r.t B:

$$s_i(\mathbf{r}) = s_i(\mathbf{e}) = \sum_{j=1}^n a_{ij} u_j(\mathbf{e})$$

since $\mathbf{h}_i = \sum_{j=1}^n a_{ij}\mathbf{b}_j$ and $\mathbf{b}_j \cdot \mathbf{e} = u_j(\mathbf{e})$.

Contructing the system

Let B be an MDS matrix with structure constants μ_l^{ij}. Define U_{ij} in the variables $U_1, \ldots, U_n$ by

$$U_{ij} = \sum_{l=1}^{n} \mu_l^{ij} U_l.$$

The ideal $J(\mathbf{r})$ in $F_q[U_1, \ldots, U_n]$ is generated by

$$\sum_{l=1}^{n} a_{jl} U_l - s_j(\mathbf{r}) \text{ for } j = 1, \ldots, n-k.$$

The ideal $I(t, U, V)$ in $F_q[U_1, \ldots, U_n, V_1, \ldots, V_t]$ is generated by

$$\sum_{j=1}^{t} U_{ij} V_j - U_{i,t+1} \text{ for } i = 1, \ldots, n$$

Let $J(t, \mathbf{r})$ be the ideal in $F_q[U_1, \ldots, U_n, V_1, \ldots, V_t]$ generated by $J(\mathbf{r})$ and $I(t, U, V)$.

Main theorem

Let B be an MDS matrix with structure constants μ_l^{ij}. Let H be a check matrix of the code C such that $H = AB$ as above. Let $\mathbf{r} = \mathbf{c} + \mathbf{e}$ be a received word with $\mathbf{c} \in C$ the codeword sent and $\mathbf{e}$ the error vector. Suppose that $wt(\mathbf{e}) \neq 0$ and $wt(\mathbf{e}) \leq \lfloor (d(C) - 1)/2 \rfloor$. Let t be the smallest positive integer such that $J(t, \mathbf{r})$ has a solution $(\mathbf{u}, \mathbf{v})$ over the algebraic closure of F_q. Then

- $wt(\mathbf{e}) = t$ and the solution is unique and of multiplicity one satisfying $\mathbf{u} = \mathbf{u}(\mathbf{e})$.
- the reduced Gröbner basis G for the ideal $J(t, \mathbf{r})$ w.r.t any monomial ordering is

  $$U_i - u_i(\mathbf{e}), i = 1, \ldots, n,$$
  $$V_j - v_j, j = 1, \ldots, t,$$

  where $(\mathbf{u}(\mathbf{e}), \mathbf{v})$ is the unique solution.

For an example see `sysQE` in Section D.10.2 [decodegb_lib], page 797. More on this method can be found in [[BP2008a]], page 718.

C.8.6 References for decoding with Groebner bases

- [ABF2002] Augot D.; Bardet M.; Faugére J.-C.: Efficient Decoding of (binary) Cyclic Codes beyond the correction capacity of the code using Gröbner bases. INRIA Report (2002) 4652
- [ABF2008] Augot D.; Bardet M.; Faugére, J.-C.: On the decoding of cyclic codes with Newton identities. to appear in Special Issue "Gröbner Bases Techniques in Cryptography and Coding Theory" of Journ. Symbolic Comp. (2008)
- [BP2008a] Bulygin S.; Pellikaan R.: Bounded distance decoding of linear error-correcting codes with Gröbner bases. to appear in Special Issue "Gröbner Bases Techniques in Cryptography and Coding Theory" of Journ. Symbolic Comp. (2008)
- [BP2008b] Bulygin S.; Pellikaan R.: Decoding and finding the minimum distance with Gröbner bases: history and new insights. to appear in "Selected topics of information and coding theory", World Scientific (2008)
- [FL1998] Fitzgerald J.; Lax R.F.: Decoding affine variety codes using Gröbner bases. Designs, Codes and Cryptography (1998) 13, 147-158
- [OS2005] Orsini E.; Sala M.: Correcting errors and erasures via the syndrome variety. J. Pure and Appl. Algebra (2005) 200, 191-226
- [S2007] Sala M.: Gröbner basis techniques to compute weight distributions of shortened cyclic codes. J. Algebra Appl. (2007) 6, 3, 403-414

C.9 References

The Centre for Computer Algebra Kaiserslautern publishes a series of preprints which are electronically available at `http://www.mathematik.uni-kl.de/~zca/Reports_on_ca`. Other sources to check are `http://symbolicnet.org/`, `http://www-sop.inria.fr/galaad/`, `http://www.bath.ac.uk/~masjpf/CAIN.html`,... and the following list of books.

For references on non-commutative algebras and algorithms, see Section 7.4.4 [References (plural)], page 363.

Text books on computational algebraic geometry

- Adams, W.; Loustaunau, P.: An Introduction to Gröbner Bases. Providence, RI, AMS, 1996
- Becker, T.; Weisspfenning, V.: Gröbner Bases - A Computational Approach to Commutative Algebra. Springer, 1993
- Cohen, H.: A Course in Computational Algebraic Number Theory, Springer, 1995
- Cox, D.; Little, J.; O'Shea, D.: Ideals, Varieties and Algorithms. Springer, 1996
- Cox, D.; Little, J.; O'Shea, D.: Using Algebraic Geometry. Springer, 1998
- Eisenbud, D.: Commutative Algebra with a View Toward Algebraic Geometry. Springer, 1995
- Greuel, G.-M.; Pfister, G.: A Singular Introduction to Commutative Algebra. Springer, 2002
- Mishra, B.: Algorithmic Algebra, Texts and Monographs in Computer Science. Springer, 1993
- Sturmfels, B.: Algorithms in Invariant Theory. Springer 1993
- Vasconcelos, W.: Computational Methods in Commutative Algebra and Algebraic Geometry. Springer, 1998

Descriptions of algorithms

- Bareiss, E.: Sylvester's identity and multistep integer-preserving Gaussian elimination. Math. Comp. 22 (1968), 565-578
- Campillo, A.: Algebroid curves in positive characteristic. SLN 813, 1980
- Chou, S.: Mechanical Geometry Theorem Proving. D.Reidel Publishing Company, 1988
- Decker, W.; Greuel, G.-M.; Pfister, G.: Primary decomposition: algorithms and comparisons. Preprint, Univ. Kaiserslautern, 1998. To appear in: Greuel, G.-M.; Matzat, B. H.; Hiss, G. (Eds.), Algorithmic Algebra and Number Theory. Springer Verlag, Heidelberg, 1998
- Decker, W.; Greuel, G.-M.; de Jong, T.; Pfister, G.: The normalisation: a new algorithm, implementation and comparisons. Preprint, Univ. Kaiserslautern, 1998
- Decker, W.; Heydtmann, A.; Schreyer, F. O.: Generating a Noetherian Normalization of the Invariant Ring of a Finite Group, 1997, to appear in Journal of Symbolic Computation
- Faugère, J. C.; Gianni, P.; Lazard, D.; Mora, T.: Efficient computation of zero-dimensional Gröbner bases by change of ordering. Journal of Symbolic Computation, 1989
- Gräbe, H.-G.: On factorized Gröbner bases, Univ. Leipzig, Inst. für Informatik, 1994
- Grassmann, H.; Greuel, G.-M.; Martin, B.; Neumann, W.; Pfister, G.; Pohl, W.; Schönemann, H.; Siebert, T.: On an implementation of standard bases and syzygies in SINGULAR. Proceedings of the Workshop Computational Methods in Lie theory in AAECC (1995)
- Greuel, G.-M.; Pfister, G.: Advances and improvements in the theory of standard bases and syzygies. Arch. d. Math. 63(1995)
- Kemper; Generating Invariant Rings of Finite Groups over Arbitrary Fields. 1996, to appear in Journal of Symbolic Computation

- Kemper and Steel: Some Algorithms in Invariant Theory of Finite Groups. 1997
- Lee, H.R.; Saunders, B.D.: Fraction Free Gaussian Elimination for Sparse Matrices. Journal of Symbolic Computation (1995) 19, 393-402
- Schönemann, H.: Algorithms in Singular, Reports on Computer Algebra 2(1996), Kaiserslautern
- Siebert, T.: On strategies and implementations for computations of free resolutions. Reports on Computer Algebra 8(1996), Kaiserslautern
- Wang, D.: Characteristic Sets and Zero Structure of Polynomial Sets. Lecture Notes, RISC Linz, 1989

Appendix D SINGULAR libraries

Singular comes with a set of standard libraries. Their content is described in the following subsections.

Use the `LIB` command (see Section 5.1.77 [LIB], page 208) for loading of single libraries, and the command `LIB "all.lib";` for loading all libraries.

See also Section 7.7 [Non-commutative libraries], page 370.

Interpreter libraries:

See Section 7.7 [Non-commutative libraries], page 370.

D.1 standard_lib

The library `standard.lib` provides extensions to the set of built-in commands and is automatically loaded during the start of Singular, unless Singular is started up with the `--no-stdlib` command line option (see Section 3.1.6 [Command line options], page 19).

Library: standard.lib

Purpose: Procedures which are always loaded at Start-up

Procedures:

`stdfglm(ideal[,ord])`
standard basis of ideal via fglm [and ordering ord]

`stdhilb(ideal[,h])`
Hilbert driven Groebner basis of ideal

`groebner(ideal,...)`
standard basis using a heuristically chosen method

`res(ideal/module,[i])`
free resolution of ideal or module

`sprintf(fmt,...)`
returns fomatted string

`fprintf(link,fmt,..)`
writes formatted string to link

`printf(fmt,...)`
displays formatted string

`weightKB(stc,dd,vl)`
degree dd part of a kbase w.r.t. some weigths

`qslimgb(i)`
computes a standard basis with slimgb in a qring

`par2varRing([i])`
create a ring making pars to vars, together with i

`datetime()`
return date and time as a string

`max(i_1,...,i_k)`
maximum of i_1, ..., i_k

`min(i_1,...,i_k)`
minimum of i_1, ..., i_k

D.1.1 qslimgb

Procedure from library `standard.lib` (see Section D.1 [standard_lib], page 721).

Usage: qslimgb(i); i ideal or module

Return: same type as input, a standard basis of i computed with slimgb

Note: Only as long as slimgb does not know qrings qslimgb should be used in case the basering is (possibly) a quotient ring.
The quotient ideal is added to the input and slimgb is applied.

Example:

```
   ring R  = (0,v),(x,y,z,u),dp;
qring Q = std(x2-y3);
ideal i = x+y2,xy+yz+zu+u*v,xyzu*v-1;
ideal j = qslimgb(i); j;
↦ j[1]=y-1
↦ j[2]=x+1
↦ j[3]=(v)*z+(v2)*u+(-v-1)
↦ j[4]=(-v2)*u2+(v+1)*u+1
module m = [x+y2,1,0], [1,1,x2+y2+xyz];
print(qslimgb(m));
↦ y2+x,x2+xy,1,        0,    0,    -x,          -xy-xz-x,
↦ 1,   y,    1,        y3-x2,0,    y2-1,        y2z-xy-x-z,
↦ 0,   0,    xyz+x2+y2,0,    y3-x2,x2y2+x3z+x2y,x3z2-x3y-2*x3-xy2
```

D.1.2 par2varRing

Procedure from library `standard.lib` (see Section D.1 [standard_lib], page 721).

Usage: par2varRing([l]); l list of ideals/modules [default:l=empty list]

Return: list, say L, with L[1] a ring where the parameters of the basering have been converted to an additional last block of variables, all of weight 1, and ordering dp.
If a list l with l[i] an ideal/module is given, then
l[i] + minpoly*freemodule(nrows(l[i])) is mapped to an ideal/module in L[1] with name Id(i).
If the basering has no parameters then L[1] is the basering.

Example:

```
   ring R = (0,x),(y,z,u,v),lp;
minpoly = x2+1;
ideal i = x3,x2+y+z+u+v,xyzuv-1; i;
↦ i[1]=(-x)
↦ i[2]=y+z+u+v-1
↦ i[3]=(x)*yzuv-1
def P = par2varRing(i)[1]; P;
↦ // coefficients: QQ
↦ // number of vars : 5
↦ //        block   1 : ordering lp
↦ //                  : names    y z u v
↦ //        block   2 : ordering dp
↦ //                  : names    x
↦ //        block   3 : ordering C
setring(P);
```

```
Id(1);
↦ Id(1)[1]=-x
↦ Id(1)[2]=y+z+u+v-1
↦ Id(1)[3]=yzuvx-1
↦ Id(1)[4]=x2+1
setring R;
module m = x3*[1,1,1], (xyzuv-1)*[1,0,1];
def Q = par2varRing(m)[1]; Q;
↦ // coefficients: QQ
↦ // number of vars : 5
↦ //        block   1 : ordering lp
↦ //                  : names    y z u v
↦ //        block   2 : ordering dp
↦ //                  : names    x
↦ //        block   3 : ordering C
setring(Q);
print(Id(1));
↦ -x,yzuvx-1,x2+1,0,   0,
↦ -x,0,      0,   x2+1,0,
↦ -x,yzuvx-1,0,   0,   x2+1
```

D.2 General purpose

D.2.1 all_lib

The library `all.lib` provides a convenient way to load all libraries of the SINGULAR distribution.

Example:

```
option(loadLib);
LIB "all.lib";
↦ // ** loaded all.lib (4.1.1.0,Jan_2018)
↦ // ** loaded ratgb.lib (4.1.1.0,Dec_2017)
↦ // ** loaded qmatrix.lib (4.1.1.0,Dec_2017)
↦ // ** loaded purityfiltration.lib (4.1.1.0,Dec_2017)
↦ // ** loaded perron.lib (4.1.1.0,Dec_2017)
↦ // ** loaded nctools.lib (4.1.1.0,Dec_2017)
↦ // ** loaded ncpreim.lib (4.1.1.0,Dec_2017)
↦ // ** loaded dmodloc.lib (4.1.1.0,Dec_2017)
↦ // ** loaded nchomolog.lib (4.1.1.0,Dec_2017)
↦ // ** loaded ncfactor.lib (4.1.1.0,Jan_2018)
↦ // ** loaded ncdecomp.lib (4.1.1.0,Dec_2017)
↦ // ** loaded ncalg.lib (4.1.1.0,Dec_2017)
↦ // ** loaded ncall.lib (4.1.1.0,Dec_2017)
↦ // ** loaded involut.lib (4.1.1.0,Dec_2017)
↦ // ** loaded gkdim.lib (4.1.1.0,Dec_2017)
↦ // ** loaded freegb.lib (4.1.1.0,Dec_2017)
↦ // ** loaded fpaprops.lib (4.1.1.0,Dec_2017)
↦ // ** loaded fpadim.lib (4.1.1.0,Dec_2017)
↦ // ** loaded dmodvar.lib (4.1.1.0,Dec_2017)
↦ // ** loaded dmodapp.lib (4.1.1.0,Dec_2017)
↦ // ** loaded dmod.lib (4.1.1.0,Dec_2017)
↦ // ** loaded central.lib (4.1.1.0,Dec_2017)
↦ // ** loaded bfun.lib (4.1.1.0,Dec_2017)
```

```
↦ // ** loaded bimodules.lib (4.1.1.0,Dec_2017)
↦ // ** loaded zeroset.lib (4.1.1.0,Dec_2017)
↦ // ** loaded weierstr.lib (4.1.1.0,Dec_2017)
↦ // ** loaded triang.lib (4.1.1.0,Dec_2017)
↦ // ** loaded toric.lib (4.1.1.0,Dec_2017)
↦ // ** loaded teachstd.lib (4.1.1.0,Dec_2017)
↦ // ** loaded surfex.lib (4.1.1.0,Dec_2017)
↦ // ** loaded surfacesignature.lib (4.1.1.0,Dec_2017)
↦ // ** loaded surf_jupyter.lib (4.1.1.0,Dec_2017)
↦ // ** loaded surf.lib (4.1.1.0,Dec_2017)
↦ // ** loaded stratify.lib (4.1.1.0,Dec_2017)
↦ // ** loaded spectrum.lib (4.1.1.0,Dec_2017)
↦ // ** loaded spcurve.lib (4.1.1.0,Dec_2017)
↦ // ** loaded solve.lib (4.1.1.0,Dec_2017)
↦ // ** loaded signcond.lib (4.1.1.0,Dec_2017)
↦ // ** loaded sing4ti2.lib (4.1.1.0,Dec_2017)
↦ // ** loaded sing.lib (4.1.1.0,Dec_2017)
↦ // ** loaded sheafcoh.lib (4.1.1.0,Dec_2017)
↦ // ** loaded sagbi.lib (4.1.1.0,Dec_2017)
↦ // ** loaded rootsur.lib (4.1.1.0,Dec_2017)
↦ // ** loaded rootsmr.lib (4.1.1.0,Dec_2017)
↦ // ** loaded rinvar.lib (4.1.1.0,Dec_2017)
↦ // ** loaded ring.lib (4.1.1.0,Dec_2017)
↦ // ** loaded reszeta.lib (4.1.1.0,Dec_2017)
↦ // ** loaded resolve.lib (4.1.1.0,Dec_2017)
↦ // ** loaded resjung.lib (4.1.1.0,Dec_2017)
↦ // ** loaded resgraph.lib (4.1.1.0,Dec_2017)
↦ // ** loaded resbinomial.lib (4.1.1.0,Dec_2017)
↦ // ** loaded reesclos.lib (4.1.1.0,Dec_2017)
↦ // ** loaded locnormal.lib (4.1.1.0,Dec_2017)
↦ // ** loaded realrad.lib (4.1.1.0,Dec_2017)
↦ // ** loaded realclassify.lib (4.1.1.0,Dec_2017)
↦ // ** loaded rootisolation.lib (4.1.1.0,Dec_2017)
↦ // ** loaded /home/hannes/singular/doc/../Singular/MOD/interval.so
↦ // ** loaded classify2.lib (4.1.1.0,Dec_2017)
↦ // ** loaded gfa.lib.so
↦ // ** loaded polyclass.lib (4.1.1.0,Dec_2017)
↦ // ** loaded random.lib (4.1.1.0,Dec_2017)
↦ // ** loaded qhmoduli.lib (4.1.1.0,Dec_2017)
↦ // ** loaded primitiv.lib (4.1.1.0,Dec_2017)
↦ // ** loaded primdecint.lib (4.1.1.0,Dec_2017)
↦ // ** loaded primdec.lib (4.1.1.0,Dec_2017)
↦ // ** loaded presolve.lib (4.1.1.0,Dec_2017)
↦ // ** loaded poly.lib (4.1.1.0,Dec_2017)
↦ // ** loaded pointid.lib (4.1.1.0,Dec_2017)
↦ // ** loaded phindex.lib (4.1.1.0,Dec_2017)
↦ // ** loaded paraplanecurves.lib (4.1.1.0,Dec_2017)
↦ // ** loaded ntsolve.lib (4.1.1.0,Dec_2017)
↦ // ** loaded normaliz.lib (4.1.1.0,Dec_2017)
↦ // ** loaded normal.lib (4.1.1.0,Dec_2017)
↦ // ** loaded curveInv.lib (4.1.1.0,Dec_2017)
↦ // ** loaded noether.lib (4.1.1.0,Dec_2017)
↦ // ** loaded mregular.lib (4.1.1.0,Dec_2017)
```

```
↦ // ** loaded mprimdec.lib (4.1.1.0,Dec_2017)
↦ // ** loaded monomialideal.lib (4.1.1.0,Dec_2017)
↦ // ** loaded mondromy.lib (4.1.1.0,Dec_2017)
↦ // ** loaded modstd.lib (4.1.1.0,Dec_2017)
↦ // ** loaded parallel.lib (4.1.1.0,Dec_2017)
↦ // ** loaded tasks.lib (4.1.1.0,Dec_2017)
↦ // ** loaded resources.lib (4.1.1.0,Dec_2017)
↦ // ** loaded modular.lib (4.1.1.0,Dec_2017)
↦ // ** loaded matrix.lib (4.1.1.0,Dec_2017)
↦ // ** loaded makedbm.lib (4.1.1.0,Dec_2017)
↦ // ** loaded linalg.lib (4.1.1.0,Dec_2017)
↦ // ** loaded latex.lib (4.1.1.0,Dec_2017)
↦ // ** loaded kskernel.lib (4.1.1.0,Dec_2017)
↦ // ** loaded jacobson.lib (4.1.1.0,Dec_2017)
↦ // ** loaded intprog.lib (4.1.1.0,Dec_2017)
↦ // ** loaded inout.lib (4.1.1.0,Dec_2017)
↦ // ** loaded integralbasis.lib (4.1.1.0,Dec_2017)
↦ // ** loaded hyperel.lib (4.1.1.0,Dec_2017)
↦ // ** loaded homolog.lib (4.1.1.0,Dec_2017)
↦ // ** loaded hnoether.lib (4.1.1.0,Dec_2017)
↦ // ** loaded grwalk.lib (4.1.1.0,Dec_2017)
↦ // ** loaded groups.lib (4.1.1.0,Dec_2017)
↦ // ** loaded grobcov.lib (4.1.1.0,Feb_2018)
↦ // ** loaded graphics.lib (4.1.1.0,Dec_2017)
↦ // ** loaded gmssing.lib (4.1.1.0,Dec_2017)
↦ // ** loaded gmspoly.lib (4.1.1.0,Dec_2017)
↦ // ** loaded general.lib (4.1.1.0,Dec_2017)
↦ // ** loaded finvar.lib (4.1.1.0,Dec_2017)
↦ // ** loaded equising.lib (4.1.1.0,Dec_2017)
↦ // ** loaded elim.lib (4.1.1.0,Dec_2017)
↦ // ** loaded deform.lib (4.1.1.0,Feb_2018)
↦ // ** loaded decodegb.lib (4.1.1.0,Dec_2017)
↦ // ** loaded curvepar.lib (4.1.1.0,Dec_2017)
↦ // ** loaded crypto.lib (4.1.1.0,Dec_2017)
↦ // ** loaded control.lib (4.1.1.0,Dec_2017)
↦ // ** loaded compregb.lib (4.1.1.0,Dec_2017)
↦ // ** loaded classify.lib (4.1.1.0,Dec_2017)
↦ // ** loaded cisimplicial.lib (4.1.1.0,Dec_2017)
↦ // ** loaded brnoeth.lib (4.1.1.0,Dec_2017)
↦ // ** loaded atkins.lib (4.1.1.0,Dec_2017)
↦ // ** loaded assprimeszerodim.lib (4.1.1.0,Dec_2017)
↦ // ** loaded arcpoint.lib (4.1.1.0,Dec_2017)
↦ // ** loaded algebra.lib (4.1.1.0,Dec_2017)
↦ // ** loaded alexpoly.lib (4.1.1.0,Dec_2017)
↦ // ** loaded aksaka.lib (4.1.1.0,Dec_2017)
↦ // ** loaded ainvar.lib (4.1.1.0,Dec_2017)
↦ // ** loaded absfact.lib (4.1.1.0,Dec_2017)
```

D.2.2 compregb_lib

Library: compregb.lib

Purpose: experimental implementation for comprehensive Groebner systems

Author: Akira Suzuki (http://kurt.scitec.kobe-u.ac.jp/~sakira/CGBusingGB/) (<sakira@kobe-u.ac.jp>)

Overview: see "A Simple Algorithm to compute Comprehensive Groebner Bases using Groebner Bases" by Akira Suzuki and Yosuke Sato for details.

Procedures:

`cgs(polys,vars,pars,R1,R2)`
: comprehensive Groebner systems

`base2str(G)`
: pretty print of the result G

D.2.3 general_lib

Library: general.lib

Purpose: Elementary Computations of General Type

Procedures:

`A_Z("a",n)`
: string a,b,... of n comma separated letters

`ASCII([n,m])`
: string of printable ASCII characters (number n to m)

`absValue(c)`
: absolute value of c

`binomial(n,m[,../..])`
: n choose m (type int), [type bigint]

`deleteSublist(iv,l)`
: delete entries given by iv from list l

`factorial(n[,../..])`
: n factorial (=n!) (type int), [type bigint]

`fibonacci(n)`
: nth Fibonacci number

`kmemory([n[,v]])`
: active [allocated] memory in kilobyte

`killall()`
: kill all user-defined variables

`number_e(n)`
: compute exp(1) up to n decimal digits

`number_pi(n)`
: compute pi (area of unit circle) up to n digits

`primes(n,m)`
: intvec of primes p, n<=p<=m

`product(../..[,v])`
: multiply components of vector/ideal/...[indices v]

`sort(ideal/module)`
: sort generators according to monomial ordering

`sum(vector/id/..[,v])`
: add components of vector/ideal/...[with indices v]

`watchdog(i,cmd)`
: only wait for result of command cmd for i seconds

`primecoeffs(J[,q])`
: primefactors <= min(p,32003) of coeffs of J

`timeStd(i,d)`
: std(i) if the standard basis computation finished after d-1 seconds and i otherwhise

`timeFactorize(p,d)`
: factorize(p) if the factorization finished after d-1 seconds otherwhise f is considered to be irreducible

`factorH(p)`
: changes variables to become the last variable the principal one in the multivariate factorization and factorizes then the polynomial

D.2.4 grobcov_lib

Library: grobcov.lib

Purpose: February 2018. Groebner Cover for parametric ideals. Comprehensive Groebner Systems, Groebner Cover, Canonical Forms, Parametric Polynomial Systems, Automatic Deduction of Geometric Theorems, Dynamic Geometry, Loci, Envelope, Constructible sets. See: A. Montes A, M. Wibmer, "Groebner Bases for Polynomial Systems with parameters", Journal of Symbolic Computation 45 (2010) 1391-1425. (https://www.mat.upc.edu//en/people/antonio.montes/).

Important:
The book, not yet published:
A. Montes. " The Groebner Cover":
(Discussing Parametric Polynomial Systems).
can be used as a user manual of all
the routines included in this library.
It defines and proves all the theoretic results used here, and shows examples of all the routines.
It will be published soon.
There are many previous papers realted to the subject, but the book actualices all the contents.

Authors: Antonio Montes (Universitat Politecnica de Catalunya), Hans Schoenemann (Technische Universitaet Kaiserslautern).

Overview: In 2010, the library was designed to contain
Montes-Wibmer's
algorithms for computing the Canonical Groebner
Cover of a parametric ideal. The central routine is grobcov. Given a parametric ideal, grobcov outputs its Canonical Groebner Cover, consisting of a set
of triplets of (lpp, basis, segment). The basis

(after normalization) is the reduced Groebner basis for each point of the segment. The segments
are disjoint, locally closed and correspond to
constant lpp (leading power product) of the basis,
and are represented in canonical representation.
The segments cover the whole parameter space.
The output is canonical, it only depends on the
given parametric ideal and the monomial order.
This is much more than a simple Comprehensive
Groebner System. The algorithm grobcov allows
options to solve partially the problem when the
whole automatic algorithm does not finish in
reasonable time.

grobcov uses a first algorithm cgsdr that outputs a disjoint reduced Comprehensive Groebner System
with constant lpp. For this purpose, in this library, the implemented algorithm is Kapur-Sun-Wang
algorithm, because it is actually the most efficient algorithm known for this purpose.
D. Kapur, Y. Sun, and D.K. Wang "A New Algorithm
for Computing Comprehensive Groebner Systems".
Proceedings of ISSAC'2010, ACM Press, (2010), 29-36.

The library has evolved to include new applications of the Groebner Cover, and new theoretical developments have been done. The actual version also includes a
routine (ConsLevels) for computing the canonical form of a constructible set, given as a union of locally closed sets. It determines the canonical locally closed level sets of a constructible set. It is described in: J.M. Brunat, A. Montes, "Computing the canonical
representation of constructible sets".
Math. Comput. Sci. (2016) 19: 165-178.

A new routine locus has been included to compute
loci of points, and determining the taxonomy of the components. It is described in the book
A. Montes. "The Groebner Cover" (Discussing
Parametric Polynomial Systems).
Additional routines to transform the output to string (locusdg, locusto) are also included and used in the Dynamic Geometry software GeoGebra. They were
described in:
M.A. Abanades, F. Botana, A. Montes, T. Recio:
"An Algebraic Taxonomy for Locus Computation in
Dynamic Geometry".
Computer-Aided Design 56 (2014) 22-33.

Recently also routines for computing the generalized envelope of a family of hypersurfaces (envelop),
to be used in Dynamic Geometry, has been included
and is described in the book
A. Montes. "The Groebner Cover" (Discussing
Parametric Polynomial Systems).

The last inclusion is an automatic algorithm for
Automatic Deduction of Geometric Theorems,
described in the book "Groebner Cover".

This version was finished on 10/02/2018

Notations: Before calling any routine of the library grobcov, the user must define the ideal Q[a][x], and all the input polynomials and ideals defined on it.
Internally the routines define and use also other
ideals: Q[a], Q[x,a] and so on.

Procedures:

`grobcov(F)`
: Is the basic routine giving the canonical Groebner Cover of the parametric ideal F. This routine accepts many options, that allow to obtain results even when the canonical computation does not finish in reasonable time.

`cgsdr(F)`
: Is the procedure for obtaining a first disjoint, reduced Comprehensive Groebner System that is used in grobcov, but can also be used independently if only a CGS is required. It is a more efficient routine than buildtree (the own routine of 2010 that is no more available). Now, Kapur-Sun-Wang (KSW) algorithm is used.

`pdivi(f,F)`
: Performs a pseudodivision of a parametric polynomial by a parametric ideal.

`pnormalf(f,E,N)`
: Reduces a parametric polynomial f over V(E) V(N). E is the null ideal and N the non-null ideal over the parameters.

`Crep(N,M)`
: Computes the canonical C-representation of V(N) V(M). It can be called in Q[a] or in Q[a][x], but the ideals N,M can only contain parameters of Q[a].

`Prep(N,M)`
: Computes the canonical P-representation of V(N) V(M). It can be called in Q[a] or in Q[a][x], but the ideals N,M can only contain parameters of Q[a].

`PtoCrep(L)`
: Starting from the canonical Prep of a locally closed set computes its Crep.

`extendpoly(f,p,q)`
: Given the generic representation f of an I-regular function F defined by poly f on V(p) V(q) it returns its full representation.

`extendGC(GC)`
: When the grobcov of an ideal has been computed with the default option ("ext",0) and the explicit option ("rep",2) (which is not the default), then one can call extendGC(GC) (and options) to obtain the full representation of the bases. With the default option ("ext",0) only the generic representation of the bases is computed, and one can obtain the full representation using extendGC.

`ConsLevels(L)`

Given a list L of locally closed sets, it returns the closures of the canonical levels of the constructible set and its complements of the union of them.

`Levels(L)`
: Transforms the output of ConsLevels into the proper Levels of the constructible set.

`locus(G)`
: Special routine for determining the geometrical locus of points verifying given conditions. Given a parametric ideal J in Q[x1,..,xn][u1,..,um] with parameters (x1,..,xn) and variables (u1,..,um), representing the system determining the n-dimensional locus with tracer point (x1,..,xn) verifying certain properties, one can apply locus to the system F, for obtaining the locus. locus provides all the components of the locus and determines their taxonomy, that can be: "Normal", "Special", "Accumulation", "Degenerate". The coordinates of a mover point, if it exist, should be placed as the n last u-variables.

`locusdg(G)`
: Is a special routine that determines the "Relevant" components of the locus in dynamic geometry. It is to be called to the output of locus and selects from it the "Normal", and"Accumulation" components.

`envelop(F,C)`
: Special routine for determining the envelop of a family of hyper-surfaces F in Q[x1,..,xn][t1,..,tm] depending on an ideal of constraints C in Q[t1,..,tm]. It computes the locus of the envelop, and detemines the different components as well as its taxonomy: "Normal", "Special", "Accumulation", "Degenerate". (See help for locus).

`locusto(L)`
: Transforms the output of locus, locusdg, envelop into a string that can be reed from different computational systems.

`stdlocus(iF)`
: Simple procedure to determine the components of the locus, alternative to locus that uses only standard GB computation. Cannot determine the taxonomy of the irreducible components.

`AssocTanToEnv(F,C,E)`
: Having computed an envelop component E of a family of hyper-surfaces F, with constraints C, it returns the parameter values of the associated tangent hyper-surface of the family passing at one point of the envelop component E.

`FamElemsAtEnvCompPoints(F,C,E)`
: Having computed an envelop component E of a family of hyper-surfaces F, with constraints C, it returns the parameter values of all the hyper-surfaces of the family passing at one point of the envelop component E.

`discrim(f,x)`
: Determines the factorized discriminant of a degree 2 polynomial in the variable x. The polynomial can be defined on any ring where x is a variable. The polynomial f can depend on parameters and variables.

`WLemma(F,A)`
: Given an ideal F in Q[a][x] and an ideal A in Q[a], it returns the list (lpp,B,S) were B is the reduced Groebner basis of the specialized F over the

segment S, subset of V(A) with top A, determined by Wibmer's Lemma. S is determined in P-representation (or optionally in C-representation). The basis is given by I-regular functions.

`intersectpar(L)`
Auxiliary routine. Given a list of ideals definend on K[a][x] it determines the intersection of all of them in K[x,a]

`ADGT(H,T,H1,T1)`
Given 4 ideals H,T,H1,T1 in Q[a][x], corresponding to a problem of Automatic Deduction of Geometric Theorems, it determines the supplementary conditions over the parameters for the Proposition (H and not H1) => (T and not T1) to be a Theorem. If H1=1 then H1 is not considered, and analogously for T1.

See also: Section D.2.2 [compregb_lib], page 725.

D.2.5 inout_lib

Library: inout.lib

Purpose: Printing and Manipulating In- and Output

Procedures:

`allprint(list)`
print list if ALLprint is defined, with pause if >0

`lprint(poly/...[,n])`
display poly/... fitting to pagewidth [size n]

`pmat(matrix[,n])`
print form-matrix [first n chars of each colum]

`rMacaulay(string)`
read Macaulay_1 output and return its Singular format

`show(any)`
display any object in a compact format

`showrecursive(id,p)`
display id recursively with respect to variables in p

`split(string,n)`
split given string into lines of length n

`tab(n)` string of n space tabs

`pause([prompt])`
stop the computation until user input

D.2.6 modular_lib

Library: modular.lib

Purpose: An abstraction layer for modular techniques

Author: Andreas Steenpass, e-mail: steenpass@mathematik.uni-kl.de

Overview: This library is an abstraction layer for modular techniques which are well-known to speed up many computations and to be easy parallelizable.
The basic idea is to execute some computation modulo several primes and then to lift the result back to characteristic zero via the farey rational map and chinese remaindering. It is thus possible to overcome the often problematic coefficient swell and to run the modular computations in parallel.
In Singular, modular techniques have been quite successfully employed for several applications. A first implementation was done for Groebner bases in Singular's Section D.4.13 [modstd_lib], page 748, a pioneering work by Stefan Steidel. Since the algorithm is basically the same for all applications, this library aims at preventing library authors from writing the same code over and over again by providing an appropriate abstraction layer. It also offers one-line commands for ordinary Singular users who want to take advantage of modular techniques for their own calculations. Thus modular techniques can be regarded as a parallel skeleton of their own.
The terminology (such as 'pTest' and 'finalTest') follows Singular's Section D.4.13 [modstd_lib], page 748 and [1].

References:

[1] Nazeran Idrees, Gerhard Pfister, Stefan Steidel: Parallelization of Modular Algorithms. Journal of Symbolic Computation 46, 672-684 (2011). http://arxiv.org/abs/1005.5663

Procedures:

`modular(...)`
: execute a command modulo several primes and lift the result back to characteristic zero

See also: Section D.4.3 [assprimeszerodim_lib], page 742; Section 4.9 [link], page 95; Section D.4.13 [modstd_lib], page 748; Section D.2.7 [parallel_lib], page 732; Section D.2.12 [tasks_lib], page 737.

D.2.7 parallel_lib

Library: parallel.lib

Purpose: An abstraction layer for parallel skeletons

Author: Andreas Steenpass, e-mail: steenpass@mathematik.uni-kl.de

Overview: This library provides implementations of several parallel 'skeletons' (i.e. ways in which parallel tasks rely upon and interact with each other). It is based on the library tasks.lib and aims at both ordinary Singular users as well as authors of Singular libraries.

Procedures:

`parallelWaitN()`
: execute several jobs in parallel and wait for N of them to finish

`parallelWaitFirst()`
: execute several jobs in parallel and wait for the first to finish

`parallelWaitAll()`
: execute several jobs in parallel and wait for all of them to finish

`parallelTestAND()`
: run several tests in parallel and determine if they all succeed

`parallelTestOR()`
: run several tests in parallel and determine if any of them succeeds

See also: Section D.4.12 [modnormal_lib], page 748; Section D.4.13 [modstd_lib], page 748; Section D.2.10 [resources_lib], page 735; Section D.2.12 [tasks_lib], page 737.

D.2.8 poly_lib

Library: poly.lib

Purpose: Procedures for Manipulating Polys, Ideals, Modules

Authors: O. Bachmann, G.-M. Greuel, A. Fruehbis

Procedures:

`cyclic(int)`
: ideal of cyclic n-roots

`elemSymmId(int)`
: ideal of elementary symmetric polynomials

`katsura([i])`
: katsura [i] ideal

`freerank(poly/...)`
: rank of coker(input) if coker is free else -1

`is_zero(poly/...)`
: int, =1 resp. =0 if coker(input) is 0 resp. not

`lcm(ideal)`
: lcm of given generators of ideal

`maxcoef(poly/...)`
: maximal length of coefficient occurring in poly/...

`maxdeg(poly/...)`
: int/intmat = degree/s of terms of maximal order

`maxdeg1(poly/...)`
: int = [weighted] maximal degree of input

`mindeg(poly/...)`
: int/intmat = degree/s of terms of minimal order

`mindeg1(poly/...)`
: int = [weighted] minimal degree of input

`normalize(poly/...)`
: normalize poly/... such that leading coefficient is 1

`rad_con(p,I)`
: check radical containment of polynomial p in ideal I

`content(f)`
: content of polynomial/vector f

`mod2id(M,iv)`
: conversion of a module M to an ideal

`id2mod(i,iv)`
: conversion inverse to mod2id

`substitute(I,...)`
: substitute in I variables by polynomials

`subrInterred(i1,i2,iv)`
: interred w.r.t. a subset of variables

`newtonDiag(f)`
: Newton diagram of a polynomial

`hilbPoly(I)`
: Hilbert polynomial of basering/I

D.2.9 random_lib

Library: random.lib

Purpose: Creating Random and Sparse Matrices, Ideals, Polys

Procedures:

`genericid(i[,p,b])`
: generic sparse linear combinations of generators of i

`randomid(id,[k,b])`
: random linear combinations of generators of id

`randommat(n,m[,id,b])`
: nxm matrix of random linear combinations of id

`sparseid(k,u[,o,p,b])`
: ideal of k random sparse poly's of degree d [u<=d<=o]

`sparsematrix(n,m,o[,.])`
: nxm sparse matrix of polynomials of degree<=o

`sparsemat(n,m[,p,b])`
: nxm sparse integer matrix with random coefficients

`sparsepoly(u[,o,p,b])`
: random sparse polynomial with terms of degree in [u,o]

`sparsetriag(n,m[,.])`
: nxm sparse lower-triag intmat with random coefficients

`sparseHomogIdeal(k,u,[,.])`
: ideal with k sparse homogeneous generators of degree in [u, o]

`triagmatrix(n,m,o[,.])`
: nxm sparse lower-triag matrix of poly's of degree<=o

`randomLast(b)`
: random transformation of the last variable

`randomBinomial(k,u,..)`
: binomial ideal, k random generators of degree >=u

D.2.10 resources_lib

Library: resources.lib

Purpose: Tools to manage the computational resources

Author: Andreas Steenpass, e-mail: steenpass@mathematik.uni-kl.de

Overview: The purpose of this library is to manage the computational resources of a Singular session. The library tasks.lib and any library build upon tasks.lib respect these settings, i.e. they will not use more computational resources than provided via resources.lib.

The provided procedures and their implementation are currently quite simple. The library can be extended later on to support, e.g., distributed computations on several servers.

Procedures:

`addcores()`
: add an integer to the number of available processor cores

`setcores()`
: set the number of available processor cores

`getcores()`
: get the number of available processor cores

`semaphore()`
: initialize a new semaphore

See also: Section D.2.7 [parallel_lib], page 732; Section D.2.12 [tasks_lib], page 737.

D.2.11 ring_lib

Library: ring.lib

Purpose: Manipulating Rings and Maps

Authors: Singular team

Procedures:

`changechar(c[,r])`
: make a copy of basering [ring r] with new char c

`changeord(o[,r])`
: make a copy of basering [ring r] with new ord o

`changevar(v[,r])`
: make a copy of basering [ring r] with new vars v

`defring("R",c,n,v,o)`
: define a ring R in specified char c, n vars v, ord o

`defrings(n[,p])`
: define ring Sn in n vars, char 32003 [p], ord ds

`defringp(n[,p])`
: define ring Pn in n vars, char 32003 [p], ord dp

`extendring("R",n,v,o)`
: extend given ring by n vars v, ord o and name it R

`fetchall(R[,str])`
: fetch all objects of ring R to basering

`imapall(R[,str])`
: imap all objects of ring R to basering

`mapall(R,i[,str])`
: map all objects of ring R via ideal i to basering

`ord_test(R)`
: test wether ordering of R is global, local or mixed

`ringtensor(s,t,..)`
: create ring, tensor product of rings s,t,...

`ringweights(r)`
: intvec of weights of ring variables of ring r

`preimageLoc(R,phi,Q)`
: computes preimage for non-global orderings

`rootofUnity(n)`
: the minimal polynomial for the n-th primitive root of unity (parameters in square brackets [] are optional)

`optionIsSet(opt)`
: check if as a string given option is set or not. hasFieldCoefficient check if the coefficient ring is considered a field hasGFCoefficient check if the coefficient ring is GF(p,k) hasZpCoefficient check if the coefficient ring is ZZ/p hasZp_aCoefficient check if the coefficient ring is an elag. ext. of ZZ/p hasQQCoefficient check if the coefficient ring is QQ

`hasNumericCoeffs(rng)`
: check for use of floating point numbers

`hasCommutativeVars(rng)`
: non-commutive or commnuative polynomial ring

`hasGlobalOrdering(rng)`
: global versus mixed/local monomial ordering

`hasMixedOrdering()`
: mixed versus global/local ordering

`hasAlgExtensionCoefficient(r)`
: coefficients are an algebraic extension

`hasTransExtensionCoefficient(r)`
: coefficients are rational functions

`isQuotientRing(rng)`
: ring is a qotient ring

`isSubModule(I,J)`
: check if I is in J as submodule

`changeordTo(r,o)`
: change the ordering of a ring to a simple one

`addvarsTo(r,vars,i)`
: add variables to a ring

`addNvarsTo(r,N,name,i)`
add N variables to a ring

D.2.12 tasks_lib

Library: tasks.lib

Purpose: A parallel framework based on tasks

Author: Andreas Steenpass, e-mail: steenpass@mathematik.uni-kl.de

Overview: This library provides a parallel framework based on tasks. It introduces a new Singular type `task`; an object of this type is a command (given by a string) applied to a list of arguments. Tasks can be computed in parallel via the procedures in this library and they can even be started recursively, i.e. from within other tasks.

tasks.lib respects the limits for computational resources defined in Section D.2.10 [resources_lib], page 735, i.e., all tasks within the same Singular session will not use more computational resources than provided via resources.lib, even if tasks are started recursively.

The Singular library Section D.2.7 [parallel_lib], page 732 provides implementations of several parallel 'skeletons' based on tasks.lib.

Procedures:

`createTask()`
create a task

`killTask()`
kill a task

`copyTask()`
copy a task

`compareTasks()`
compare two tasks

`printTask()`
print a task

`startTasks()`
start tasks

`stopTask()`
stop a task

`waitTasks()`
wait for a certain number of tasks

`waitAllTasks()`
wait for all tasks

`pollTask()`
poll a task

`getCommand()`
get the command of a task

`getArguments()`
get the arguments of a task

`getResult()`
: get the result of a task

`getState()`
: get the state of a task

See also: Section D.2.7 [parallel_lib], page 732; Section D.2.10 [resources_lib], page 735.

D.3 Linear algebra

D.3.1 matrix_lib

Library: matrix.lib

Purpose: Elementary Matrix Operations

Procedures:

`compress(A)`
: matrix, zero columns from A deleted

`concat(A1,A2,..)`
: matrix, concatenation of matrices A1,A2,...

`diag(p,n)`
: matrix, nxn diagonal matrix with entries poly p

`dsum(A1,A2,..)`
: matrix, direct sum of matrices A1,A2,...

`flatten(A)`
: ideal, generated by entries of matrix A

`genericmat(n,m[,id])`
: generic nxm matrix [entries from id]

`is_complex(c)`
: 1 if list c is a complex, 0 if not

`outer(A,B)`
: matrix, outer product of matrices A and B

`power(A,n)`
: matrix/intmat, n-th power of matrix/intmat A

`skewmat(n[,id])`
: generic skew-symmetric nxn matrix [entries from id]

`submat(A,r,c)`
: submatrix of A with rows/cols specified by intvec r/c

`symmat(n[,id])`
: generic symmetric nxn matrix [entries from id]

`tensor(A,B)`
: matrix, tensor product of matrices A nd B

`unitmat(n)`
: unit square matrix of size n

`gauss_col(A)`
transform a matrix into col-reduced Gauss normal form

`gauss_row(A)`
transform a matrix into row-reduced Gauss normal form

`addcol(A,c1,p,c2)`
add p*(c1-th col) to c2-th column of matrix A, p poly

`addrow(A,r1,p,r2)`
add p*(r1-th row) to r2-th row of matrix A, p poly

`multcol(A,c,p)`
multiply c-th column of A with poly p

`multrow(A,r,p)`
multiply r-th row of A with poly p

`permcol(A,i,j)`
permute i-th and j-th columns

`permrow(A,i,j)`
permute i-th and j-th rows

`rowred(A[,any])`
reduction of matrix A with elementary row-operations

`colred(A[,any])`
reduction of matrix A with elementary col-operations

`linear_relations(E)`
find linear relations between homogeneous vectors

`rm_unitrow(A)`
remove unit rows and associated columns of A

`rm_unitcol(A)`
remove unit columns and associated rows of A

`headStand(A)`
A[n-i+1,m-j+1]:=A[i,j]

`symmetricBasis(n,k[,s])`
basis of k-th symmetric power of n-dim v.space

`exteriorBasis(n,k[,s])`
basis of k-th exterior power of n-dim v.space

`symmetricPower(A,k)`
k-th symmetric power of a module/matrix A

`exteriorPower(A,k)`
k-th exterior power of a module/matrix A

D.3.2 linalg_lib

Library: linalg.lib

Purpose: Algorithmic Linear Algebra

Authors: Ivor Saynisch (ivs@math.tu-cottbus.de)
Mathias Schulze (mschulze@mathematik.uni-kl.de)

Procedures:

`inverse(A)`
: matrix, the inverse of A

`inverse_B(A)`
: list(matrix Inv,poly p),Inv*A=p*En (using busadj(A))

`inverse_L(A)`
: list(matrix Inv,poly p),Inv*A=p*En (using lift)

`sym_gauss(A)`
: symmetric gaussian algorithm

`orthogonalize(A)`
: Gram-Schmidt orthogonalization

`diag_test(A)`
: test whether A can be diagnolized

`busadj(A)`
: coefficients of Adj(E*t-A) and coefficients of det(E*t-A)

`charpoly(A,v)`
: characteristic polynomial of A (using busadj(A))

`adjoint(A)`
: adjoint of A (using busadj(A))

`det_B(A)` determinant of A (using busadj(A))

`gaussred(A)`
: gaussian reduction: P*A=U*S, S a row reduced form of A

`gaussred_pivot(A)`
: gaussian reduction: P*A=U*S, uses row pivoting

`gauss_nf(A)`
: gaussian normal form of A

`mat_rk(A)`
: rank of constant matrix A

`U_D_O(A)` P*A=U*D*O, P,D,U,O=permutaion,diag,lower-,upper-triang

`pos_def(A,i)`
: test symmetric matrix for positive definiteness

`hessenberg(M)`
: Hessenberg form of M

`eigenvals(M)`
: eigenvalues with multiplicities of M

`minipoly(M)`
: minimal polynomial of M

`spnf(sp)` normal form of spectrum sp

`spprint(sp)`
: print spectrum sp

`jordan(M)`
: Jordan data of M

`jordanbasis(M)`
: Jordan basis and weight filtration of M

`jordanmatrix(jd)`
: Jordan matrix with Jordan data jd

`jordannf(M)`
: Jordan normal form of M

D.4 Commutative algebra

D.4.1 absfact_lib

Library: absfact.lib

Purpose: Absolute factorization for characteristic 0

Authors: Wolfram Decker, decker at math.uni-sb.de
Gregoire Lecerf, lecerf at math.uvsq.fr
Gerhard Pfister, pfister at mathematik.uni-kl.de Martin Lee, mlee at mathematik.uni-kl.de

Overview: A library for computing the absolute factorization of multivariate polynomials f with coefficients in a field K of characteristic zero. Using Trager's idea, the implemented algorithm computes an absolutely irreducible factor by factorizing over some finite extension field L (which is chosen such that V(f) has a smooth point with coordinates in L). Then a minimal extension field is determined making use of the Rothstein-Trager partial fraction decomposition algorithm. absFactorizeBCG uses the algorithm of Bertone, Cheze and Galligo for bivariate polynomials and similar ideas as above to reduce to this case.

References:

G. Cheze, G. Lecerf: Lifting and recombination techniques for absolute factorization. Journal of Complexity, 23(3):380-420, 2007. C. Bertone, G. Cheze, and A. Galligo: Modular las vegas algorithms for polynomial absolute factorization. J. Symb. Comput., 45(12):1280-1295, December 2010

Procedures:

`absFactorize()`
: absolute factorization of poly

`absFactorizeBCG()`
: absolute factorization of poly

See also: Section 5.1.35 [factorize], page 178.

D.4.2 algebra_lib

Library: algebra.lib

Purpose: Compute with Algbras and Algebra Maps

Authors: Gert-Martin Greuel, greuel@mathematik.uni-kl.de,
Agnes Eileen Heydtmann, agnes@math.uni-sb.de,
Gerhard Pfister, pfister@mathematik.uni-kl.de

Procedures:

`algebra_containment()`
query of algebra containment

`module_containment()`
query of module containment over a subalgebra

`inSubring(p,I)`
test whether polynomial p is in subring generated by I

`algDependent(I)`
computes algebraic relations between generators of I

`alg_kernel(phi)`
computes the kernel of the ringmap phi

`is_injective(phi)`
test for injectivity of ringmap phi

`is_surjective(phi)`
test for surjectivity of ringmap phi

`is_bijective(phi)`
test for bijectivity of ring map phi

`noetherNormal(id)`
noether normalization of ideal id

`mapIsFinite(R,phi,I)`
query for finiteness of map phi:R –> basering/I

`finitenessTest(i,z)`
find variables which occur as pure power in lead(i)

`nonZeroEntry(id)`
list describing non-zero entries of an identifier

D.4.3 assprimeszerodim_lib

Library: assprimeszerodim.lib

Purpose: associated primes of a zero-dimensional ideal

Authors: N. Idrees nazeranjawwad@gmail.com
G. Pfister pfister@mathematik.uni-kl.de
A. Steenpass steenpass@mathematik.uni-kl.de
S. Steidel steidel@mathematik.uni-kl.de

Overview: A library for computing the associated primes and the radical of a zero-dimensional ideal in the polynomial ring over the rational numbers, Q[x_1,...,x_n], using modular computations.

Procedures:

`zeroRadical(I)`
computes the radical of I

`assPrimes(I)`
computes the associated primes of I

See also: Section D.4.21 [primdec_lib], page 756.

D.4.4 cisimplicial_lib

Library: cisimplicial.lib

Purpose: . Determines if the toric ideal of a simplicial toric variety is a complete intersection

Authors: I.Bermejo, ibermejo@ull.es
I.Garcia-Marco, iggarcia@ull.es

Overview: A library for determining if a simplicial toric ideal is a complete intersection with NO NEED of computing explicitly a system of generators of such ideal. The procedures are based on two papers: I. Bermejo, I. Garcia-Marco and J.J. Salazar-Gonzalez: 'An algorithm for checking whether the toric ideal of an affine monomial curve is a complete intersection', J. Symbolic Computation 42 (2007) pags: 971–991 and I.Bermejo and I. Garcia-Marco: 'Complete intersections in simplicial toric varieties', Preprint (2010)

Procedures:

`minMult(a,b)`
computes the minimum multiple of a that belongs to the semigroup generated by b

`belongSemigroup(v,A[,n])`
checks whether A*x = v has a nonnegative integral solution

`oneDimBelongSemigroup(n,v[,m])`
checks whether v*x = n has a nonnegative integral solution

`cardGroup(A)`
computes the cardinal of Z^m / ZA

`isCI(A)` checks wether I(A) is a complete intersection

D.4.5 elim_lib

Library: elim.lib

Purpose: Elimination, Saturation and Blowing up

Procedures:

`blowup0(j[,s1,s2])`
create presentation of blownup ring of ideal j

`elimRing(p)`
create ring with block ordering for elimating vars in p

`elim(id,..)`
variables .. eliminated from id (ideal/module)

`elim1(id,p)`
variables .. eliminated from id (different algorithm)

`elim2(id,..)`
variables .. eliminated from id (different algorithm)

`nselect(id,v)`
select generators not containing variables given by v

`sat(id,j)`
saturated quotient of ideal/module id by ideal j

`select(id,v)`
: select generators containing all variables given by v

`select1(id,v)`
: select generators containing one variable given by v

D.4.6 ellipticcovers_lib

Library: ellipticCovers.lib

Purpose: Gromov-Witten numbers of elliptic curves

Authors: J. Boehm, boehm @ mathematik.uni-kl.de
A. Buchholz, buchholz @ math.uni-sb.de
H. Markwig hannah @ math.uni-sb.de

Overview: We implement a formula for computing the number of covers of elliptic curves. It has beed obtained by proving mirror symmetry
for arbitrary genus by tropical methods in [BBM]. A Feynman graph of genus g is a trivalent, connected graph of genus g (with 2g-2 vertices and 3g-3 edges). The branch type b=(b_1,...,b_(3g-3)) of a stable map is the multiplicity of the the edge i over a fixed base point.

Given a Feynman graph G and a branch type b, we obtain the number N_(G,b) of stable maps of branch type b from a genus g curve of topological type G to the elliptic curve by computing a path integral
over a rational function. The path integral is computed as a residue.

The sum of N_(G,b) over all branch types b of sum d gives N_(G,d)*|Aut(G)|, with the Gromov-Witten invariant N_(G,d) of degree d stable maps from a genus g curve of topological type G to the elliptic curve.

The sum of N_(G,d) over all such graphs gives the usual Gromov-Witten invariant N_(g,d) of degree d stable maps from a genus g curve to the elliptic curve.

The key function computing the numbers N_(G,b) and N_(G,d) is gromovWitten.

References:

[BBM] J. Boehm, A. Buchholz, H. Markwig: Tropical mirror symmetry for elliptic curves, arXiv:1309.5893 (2013).

Types: graph

Procedures:

`makeGraph(list, list)`
: generate a graph from a list of vertices and a lsit of edges

`printGraph(graph)`
: print procedure for graphs

`propagator(list,int)`
: propagator factor of degree d in the quotient of two variables, or propagator for fixed graph and branch type

`computeConstant(number, number)`
: constant coefficient in the Laurent series expansion of a rational function in a given variable

`evalutateIntegral(number, list)`
: path integral for a given propagator and ordered sequence of variables

`gromovWitten(number)`
: sum of path integrals for a given propagator over all orderings of the variables, or Gromov Witten invariant for a given graph and a fixed branch type, or list of Gromov Witten invariants for a given graph and all branch types

`computeGromovWitten(graph, int, int)`
: compute the Gromov Witten invariants for a given graph and some branch types generatingFunction (graph, int) multivariate generating function for the Gromov Witten invariants of a graph up to fixed degree

`partitions(int, int)`
: partitions of an integer into a fixed number of summands

`permute(list)`
: all permutations of a list

`lsum(list)`
: sum of the elements of a list

D.4.7 grwalk_lib

Library: grwalk.lib

Purpose: Groebner Walk Conversion Algorithms

Author: I Made Sulandra

Procedures:

`fwalk(ideal[,intvec])`
: standard basis of ideal via fractalwalk alg

`twalk(ideal[,intvec])`
: standard basis of ideal via Tran's alg

`awalk1(ideal[,intvec])`
: standard basis of ideal via the first alt. alg

`awalk2(ideal[,intvec])`
: standard basis of ideal via the second alt. alg

`pwalk(ideal[,intvec])`
: standard basis of ideal via perturbation walk alg

`gwalk(ideal[,intvec])`
: standard basis of ideal via groebnerwalk alg

See also: Section D.15.34 [rwalk_lib], page 864; Section D.15.36 [swalk_lib], page 866.

D.4.8 homolog_lib

Library: homolog.lib

Purpose: Procedures for Homological Algebra

Authors: Gert-Martin Greuel, greuel@mathematik.uni-kl.de,
Bernd Martin, martin@math.tu-cottbus.de
Christoph Lossen, lossen@mathematik.uni-kl.de

Procedures:

`canonMap(id)`
: the kernel and the cokernel of the canonical map

`cup(M)`
: cup: Ext^1(M',M') x Ext^1() –> Ext^2()

`cupproduct(M,N,P,p,q)`
: cup: Ext^p(M',N') x Ext^q(N',P') –> Ext^p+q(M',P')

`depth(I,M)`
: depth(I,M'), I ideal, M module, M'=coker(M)

`Ext_R(k,M)`
: Ext^k(M',R), M module, R basering, M'=coker(M)

`Ext(k,M,N)`
: Ext^k(M',N'), M,N modules, M'=coker(M), N'=coker(N)

`fitting(M,n)`
: n-th Fitting ideal of M'=coker(M), M module, n int

`flatteningStrat(M)`
: Flattening stratification of M'=coker(M), M module

`Hom(M,N)`
: Hom(M',N'), M,N modules, M'=coker(M), N'=coker(N)

`homology(A,B,M,N)`
: ker(B)/im(A), homology of complex R^k–A->M'–B->N'

`isCM(M)`
: test if coker(M) is Cohen-Macaulay, M module

`isFlat(M)`
: test if coker(M) is flat, M module

`isLocallyFree(M,r)`
: test if coker(M) is locally free of constant rank r

`isReg(I,M)`
: test if I is coker(M)-sequence, I ideal, M module

`hom_kernel(A,M,N)`
: ker(M'–A->N') M,N modules, A matrix

`kohom(A,k)`
: Hom(R^k,A), A matrix over basering R

`kontrahom(A,k)`
: Hom(A,R^k), A matrix over basering R

`KoszulHomology(I,M,n)`
: n-th Koszul homology H_n(I,coker(M)), I=ideal

`tensorMod(M,N)`
: Tensor product of modules M'=coker(M), N'=coker(N)

`Tor(k,M,N)`
: Tor_k(M',N'), M,N modules, M'=coker(M), N'=coker(N)

D.4.9 integralbasis_lib

Library: integralbasis.lib

Purpose: Integral basis in algebraic function fields

Authors: J. Boehm, j.boehm at mx.uni-saarland.de
W. Decker, decker at mathematik.uni-kl.de>
S. Laplagne, slaplagn at dm.uba.ar
F. Seelisch, seelisch at mathematik.uni-kl.de

Overview: Given an irreducible polynomial f in two variables defining a plane curve, this library implements an algorithm to compute an integral basis of the integral closure of the affine coordinate ring in the algebraic function field via normalization.
The user can choose whether the algorithm will do the computation globally or (this is the default) compute in the localization at each component of the singular locus and put everything together.

Procedures:

`integralBasis(f, intVar)`
Integral basis of an algebraic function field

D.4.10 intprog_lib

Library: intprog.lib

Purpose: Integer Programming with Groebner Basis Methods

Author: Christine Theis, email: ctheis@math.uni-sb.de

Procedures:

`solve_IP(..)`
procedures for solving Integer Programming problems

D.4.11 locnormal_lib

Library: locnormal.lib

Purpose: Normalization of affine domains using local methods

Authors: J. Boehm boehm@mathematik.uni-kl.de
W. Decker decker@mathematik.uni-kl.de
S. Laplagne slaplagn@dm.uba.ar
G. Pfister pfister@mathematik.uni-kl.de
S. Steidel steidel@mathematik.uni-kl.de
A. Steenpass steenpass@mathematik.uni-kl.de

Overview: Suppose A is an affine domain over a perfect field.
This library implements a local-to-global strategy for finding the normalization of A. Following [1], the idea is to stratify the singular locus of A, apply the normalization algorithm given in [2] locally at each stratum, and put the local results together. This approach is inherently parallel.
Furthermore we allow for the optional modular computation of the local results as provided by modnormal.lib. See again [1] for details.

References:

[1] Janko Boehm, Wolfram Decker, Santiago Laplagne, Gerhard Pfister, Stefan Steidel, Andreas Steenpass: Parallel algorithms for normalization, http://arxiv.org/abs/1110.4299, 2011.

[2] Gert-Martin Greuel, Santiago Laplagne, Frank Seelisch: Normalization of Rings, Journal of Symbolic Computation 9 (2010), p. 887-901

Procedures:

`locNormal(I, [...])`
normalization of R/I using local methods

See also: Section D.4.12 [modnormal_lib], page 748; Section D.4.18 [normal_lib], page 753.

D.4.12 modnormal_lib

Library: modnormal.lib

Purpose: Normalization of affine domains using modular methods

Authors: J. Boehm boehm@mathematik.uni-kl.de
W. Decker decker@mathematik.uni-kl.de
S. Laplagne slaplagn@dm.uba.ar
G. Pfister pfister@mathematik.uni-kl.de
A. Steenpass steenpass@mathematik.uni-kl.de
S. Steidel steidel@mathematik.uni-kl.de

Overview: Suppose A is an affine domain over a perfect field.
This library implements a modular strategy for finding the normalization of A. Following [1], the idea is to apply the normalization algorithm given in [2] over finite fields and lift the results via Chinese remaindering and rational reconstruction as described in [3]. This approch is inherently parallel.
The strategy is available both as a randomized and as a verified algorithm.

References:

[1] Janko Boehm, Wolfram Decker, Santiago Laplagne, Gerhard Pfister, Stefan Steidel, Andreas Steenpass: Parallel algorithms for normalization, preprint, 2011.

[2] Gert-Martin Greuel, Santiago Laplagne, Frank Seelisch: Normalization of Rings, Journal of Symbolic Computation 9 (2010), p. 887-901

[3] Janko Boehm, Wolfram Decker, Claus Fieker, Gerhard Pfister: The use of Bad Primes in Rational Reconstruction, preprint, 2012.

Procedures:

`modNormal(I)`
normalization of R/I using modular methods

See also: Section D.4.11 [locnormal_lib], page 747; Section D.4.18 [normal_lib], page 753.

D.4.13 modstd_lib

Library: modstd.lib

Purpose: Groebner bases of ideals using modular methods

Authors: A. Hashemi Amir.Hashemi@lip6.fr
G. Pfister pfister@mathematik.uni-kl.de
H. Schoenemann hannes@mathematik.uni-kl.de
A. Steenpass steenpass@mathematik.uni-kl.de
S. Steidel steidel@mathematik.uni-kl.de

Overview: A library for computing Groebner bases of ideals in the polynomial ring over the rational numbers using modular methods.

References:

E. A. Arnold: Modular algorithms for computing Groebner bases. J. Symb. Comp. 35, 403-419 (2003).

N. Idrees, G. Pfister, S. Steidel: Parallelization of Modular Algorithms. J. Symb. Comp. 46, 672-684 (2011).

Procedures:

`modStd(I)`
standard basis of I using modular methods

`modSyz(I)`
syzygy module of I using modular methods

`modIntersect(I,J)`
intersection of I and J using modular methods

D.4.14 monomialideal_lib

Library: monomialideal.lib

Purpose: Primary and irreducible decompositions of monomial ideals

Authors: I.Bermejo, ibermejo@ull.es
E.Garcia-Llorente, evgarcia@ull.es
Ph.Gimenez, pgimenez@agt.uva.es

Overview: A library for computing a primary and the irreducible decompositions of a monomial ideal using several methods.
In this library we also take advantage of the fact that the ideal is monomial to make some computations that are Grobner free in this case (radical, intersection, quotient...).

Procedures:

`isMonomial(id)`
checks whether an ideal id is monomial

`minbaseMon(id)`
computes the minimal monomial generating set of a monomial ideal id

`gcdMon(f,g)`
computes the gcd of two monomials f, g

`lcmMon(f,g)`
computes the lcm of two monomials f, g

`membershipMon(f,id)`
checks whether a polynomial f belongs to a monomial ideal id

`intersectMon(id1,id2)`
intersection of monomial ideals id1 and id2

`quotientMon(id1,id2)`
: quotient ideal id1:id2

`radicalMon(id)`
: computes the radical of a monomial ideal id

`isprimeMon(id)`
: checks whether a monomial ideal id is prime

`isprimaryMon(id)`
: checks whether a monomial ideal id is primary

`isirreducibleMon(id)`
: checks whether a monomial ideal id is irreducible

`isartinianMon(id)`
: checks whether a monomial ideal id is artininan

`isgenericMon(id)`
: checks whether a monomial ideal id is generic

`dimMon(id)`
: dimension of a monomial ideal id

`irreddecMon(id,..)`
: computes the irreducible decomposition of a monomial ideal id

`primdecMon(id,..)`
: computes a minimal primary decomposition of a monomial ideal id

D.4.15 mprimdec_lib

Library: mprimdec.lib

Purpose: procedures for primary decomposition of modules

Authors: Alexander Dreyer, dreyer@mathematik.uni-kl.de; adreyer@web.de

Overview: Algorithms for primary decomposition for modules based on the algorithms of Gianni, Trager and Zacharias and
Shimoyama and Yokoyama (generalization of the latter suggested by Hans-Gert Graebe, Leipzig)
using elments of primdec.lib

Remark: These procedures are implemented to be used in characteristic 0.
They also work in positive characteristic >> 0.
In small characteristic and for algebraic extensions, the procedures via Gianni, Trager, Zacharias may not terminate.

Procedures:

`separator(l)`
: computes a list of separators of prime ideals

`PrimdecA(N[,i])`
: (not necessarily minimal) primary decomposition via Shimoyama/Yokoyama (suggested by Graebe)

`PrimdecB(N,p)`
: (not necessarily minimal) primary decomposition for pseudo-primary ideals

`modDec(N[,i])`
: minimal primary decomposition via Shimoyama/Yokoyama (suggested by Graebe)

`zeroMod(N[,check])`
: minimal zero-dimensional primary decomposition via Gianni, Trager and Zacharias

`GTZmod(N[,check])`
: minimal primary decomposition via Gianni, Trager and Zacharias

`dec1var(N[,check[,ann]])`
: primary decomposition for one variable

`annil(N)` the annihilator of M/N in the basering

`splitting(N[,check[,ann]])`
: splitting to simpler modules

`primTest(i[,p])`
: tests whether i is prime or homogeneous

`preComp(N,check[,ann])`
: enhanced Version of splitting

`indSet(i)`
: lists with varstrings of(in)dependend variables

`GTZopt(N[,check[,ann]])`
: a faster version of GTZmod

`zeroOpt(N[,check[,ann]])`
: a faster version of zeroMod

D.4.16 mregular_lib

Library: mregular.lib

Purpose: Castelnuovo-Mumford regularity of homogeneous ideals

Authors: I.Bermejo, ibermejo@ull.es
Ph.Gimenez, pgimenez@agt.uva.es
G.-M.Greuel, greuel@mathematik.uni-kl.de

Overview: A library for computing the Castelnuovo-Mumford regularity of a homogeneous ideal that DOES NOT require the computation of a minimal graded free resolution of the ideal.
It also determines depth(basering/ideal) and satiety(ideal). The procedures are based on 3 papers by Isabel Bermejo and Philippe Gimenez: 'On Castelnuovo-Mumford regularity of projective curves' Proc.Amer.Math.Soc. 128(5) (2000), 'Computing the Castelnuovo-Mumford regularity of some subschemes of Pn using quotients of monomial ideals', Proceedings of MEGA-2000, J. Pure Appl. Algebra 164 (2001), and 'Saturation and Castelnuovo-Mumford regularity', Preprint (2004).

Procedures:

`regIdeal(id,[,e])`
: regularity of homogeneous ideal id

`depthIdeal(id,[,e])`
depth of S/id with S=basering, id homogeneous ideal

`satiety(id,[,e])`
saturation index of homogeneous ideal id

`regMonCurve(li)`
regularity of projective monomial curve defined by li

`NoetherPosition(id)`
Noether normalization of ideal id

`is_NP(id)`
checks whether variables are in Noether position

`is_nested(id)`
checks whether monomial ideal id is of nested type

D.4.17 noether_lib

Library: noether.lib

Purpose: Noether normalization of an ideal (not nessecary homogeneous)

Authors: A. Hashemi, Amir.Hashemi@lip6.fr

Overview: A library for computing the Noether normalization of an ideal that DOES NOT require the computation of the dimension of the ideal. It checks whether an ideal is in Noether position. A modular version of these algorithms is also provided.
The procedures are based on a paper of Amir Hashemi 'Efficient Algorithms for Computing Noether Normalization' (presented in ASCM 2007)

This library computes also Castelnuovo-Mumford regularity and satiety of an ideal. A modular version of these algorithms is also provided. The procedures are based on a paper of Amir Hashemi 'Computation of Castelnuovo-Mumford regularity and satiety' (preprint 2008)

Procedures:

`NPos_test(id)`
checks whether monomial ideal id is in Noether position

`modNpos_test(id)`
the same as above using modular methods

`NPos(id)` Noether normalization of ideal id

`modNPos(id)`
Noether normalization of ideal id by modular methods

`nsatiety(id)`
Satiety of ideal id

`modsatiety(id)`
Satiety of ideal id by modular methods

`regCM(id)`
Castelnuovo-Mumford regularity of ideal id

`modregCM(id)`
Castelnuovo-Mumford regularity of ideal id by modular methods

D.4.18 normal_lib

Library: normal.lib

Purpose: Normalization of Affine Rings

Authors: G.-M. Greuel, greuel@mathematik.uni-kl.de,
S. Laplagne, slaplagn@dm.uba.ar,
G. Pfister, pfister@mathematik.uni-kl.de
Peter Chini, chini@rhrk.uni-kl.de (normalConductor)

Procedures:

`normal(I,[...])`
normalization of an affine ring

`normalP(I,[...])`
normalization of an affine ring in positive characteristic

`normalC(I,[...])`
normalization of an affine ring through a chain of rings

`HomJJ(L)` presentation of End_R(J) as affine ring, J an ideal

`genus(I)` computes the geometric genus of a projective curve

`primeClosure(L)`
integral closure of R/p, p a prime ideal

`closureFrac(L)`
writes a poly in integral closure as element of Quot(R/p)

`iMult(L)` intersection multiplicity of the ideals of the list L

`deltaLoc(f,S)`
sum of delta invariants at conjugated singular points

`locAtZero(I)`
checks whether the zero set of I is located at 0

`norTest(I,nor)`
checks the output of normal, normalP, normalC

`getSmallest(J)`
computes the polynomial of smallest degree of J

`getOneVar(J, vari)`
computes a polynomial of J in the variable vari

`changeDenominator(U1, c1, c2, I)`
computes ideal U2 such that 1/c1*U1=1/c2*U2

`normalConductor(ideal)`
computation of the conductor as ideal in the basering

See also: Section D.4.11 [locnormal_lib], page 747; Section D.4.12 [modnormal_lib], page 748.

D.4.19 normaliz_lib

Library: normaliz.lib

Purpose: Provides an interface for the use of Normaliz 2.11 or newer within SINGULAR.

Authors: Winfried Bruns, Winfried.Bruns@Uni-Osnabrueck.de
Christof Soeger, Christof.Soeger@Uni-Osnabrueck.de

Overview: The library normaliz.lib provides an interface for the use of Normaliz 2.11 or newer within SINGULAR. The exchange of data is via files. In addition to the top level functions that aim at objects of type ideal or ring, several other auxiliary functions allow the user to apply Normaliz to data of type intmat. Therefore SINGULAR can be used as a comfortable environment for the work with Normaliz.
Please see the `Normaliz.pdf` (included in the Normaliz distribution) for a more extensive documentation of Normaliz.

Normaliz allows the use of a grading. In the Singular functions that access Normaliz the parameter grading is an intvec that assigns a (not necessarily positive) degree to every variable of the ambient polynomial ring. But it must give positive degrees to the generators given to function.

Singular and Normaliz exchange data via files. These files are automatically created and erased behind the scenes. As long as one wants to use only the ring-theoretic functions there is no need for file management.
Note that the numerical invariants computed by Normaliz can be accessed in this "automatic file mode".
However, if Singular is used as a frontend for Normaliz or the user wants to inspect data not automatically returned to Singular, then an explicit filename and a path can be specified for the exchange of data. Moreover, the library provides functions for access to these files. Deletion of the files is left to the user.

Use of this library requires the program Normaliz to be installed. You can download it from `http://www.mathematik.uni-osnabrueck.de/normaliz/`. Please make sure that the executables are in the search path or use setNmzExecPath ([setNmzExecPath], page 755).

Procedures:

`intclToricRing(ideal I)`
: computes the integral closure of the toric ring generated by the leading monomials of the elements of I in the basering

`normalToricRing(ideal I)`
: computes the normalization of the toric ring generated by the leading monomials of the elements of I

`normalToricRingFromBinomials(ideal I)`
: computes the normalization of the polynomial ring modulo the unique minimal binomial prime ideal of the binomial ideal I

`ehrhartRing(ideal I)`
: considers the exponent vectors of the elements of I as points of a lattice polytope and computes the integral cloure of the polytopal algebra

`intclMonIdeal(ideal I)`
: Computes the integral closure of the Rees algebra of the ideal generated by the leading monomials of the elements of I

`torusInvariants(intmat T)`
: computes the ring of invariants of a torus action

`finiteDiagInvariants(intmat C)`
: computes the ring of invariants of a finite abelian group acting diagonally on a polynomial ring

`diagInvariants(intmat C)`
computes the ring of invariants of a diagonalizable group

`intersectionValRings(intmat V)`
computes the intersection of the polynomial ring with the valuation rings of monomial valuations

`intersectionValRingIdeals(intmat V)`
computes ideals of monomial valuations

`showNuminvs()`
prints the numerical invariants found by Normaliz

`exportNuminvs()`
exports the numerical invariants found by Normaliz

`setNmzOption(string s, int onoff)`
sets the option s to onoff

`showNmzOptions()`
prints the enabled options to the standard output

`normaliz(intmat sgr,int nmz_mode)`
applies Normaliz

`setNmzExecPath(string nmz_exec_path_name)`
sets the path to the Normaliz executable

`writeNmzData(intmat sgr, int n_mode)`
creates an input file for Normaliz

`readNmzData(string nmz_suffix)`
reads the Normaliz output file with the specified suffix

`setNmzFilename(string nmz_filename_name)`
sets the filename for the exchange of data

`setNmzDataPath(string nmz_data_path_name)`
sets the directory for the exchange of data

`writeNmzPaths()`
writes the path names into two files

`startNmz()`
retrieves the path names written by writeNmzPaths

`rmNmzFiles()`
removes the files created for and by Normaliz

`mons2intmat(ideal I)`
returns the intmat whose rows represent the leading exponents of the elements of I

`intmat2mons(intmat expo_vecs)`
returns the ideal generated by the monomials which have the rows of expo_vecs as exponent vector

`binomials2intmat(ideal I)`
returns the intmat whose rows represent the exponents of the elements of the binomial ideal I

D.4.20 pointid_lib

Library: pointid.lib

Purpose: Procedures for computing a factorized lex GB of the vanishing ideal of a set of points via the Axis-of-Evil Theorem (M.G. Marinari, T. Mora)

Author: Stefan Steidel, steidel@mathematik.uni-kl.de

Overview: The algorithm of Cerlienco-Mureddu [Marinari M.G., Mora T., A remark on a remark by Macaulay or Enhancing Lazard Structural Theorem. Bull. of the Iranian Math. Soc., 29 (2003), 103-145] associates to each ordered set of points A:={a1,...,as} in K^n, ai:=(ai1,...,ain)
- a set of monomials N and
- a bijection phi: A –> N.
Here I(A):={f in K[x(1),...,x(n)] | f(ai)=0, for all 1<=i<=s} denotes the vanishing ideal of A and N = Mon(x(1),...,x(n)) {LM(f)|f in I(A)} is the set of monomials which do not lie in the leading ideal of I(A) (w.r.t. the lexicographical ordering with x(n)>...>x(1)). N is also called the set of non-monomials of I(A). NOTE: #A = #N and N is a monomial basis of K[x(1..n)]/I(A). In particular, this allows to deduce the set of corner-monomials, i.e. the minimal basis M:={m1,...,mr}, m1<...<mr, of its associated monomial ideal M(I(A)), such that
M(I(A))= {k*mi | k in Mon(x(1),...,x(n)), mi in M},
and (by interpolation) the unique reduced lexicographical Groebner basis G := {f1,...,fr} such that LM(fi)=mi for each i, that is, I(A)=<G>. Moreover, a variation of this algorithm allows to deduce a canonical linear factorization of each element of such a Groebner basis in the sense ot the Axis-of-Evil Theorem by M.G. Marinari and T. Mora. More precisely, a combinatorial algorithm and interpolation allow to deduce polynomials

y_mdi = x(m) - g_mdi(x(1),...,x(m-1)),
i=1,...,r; m=1,...,n; d in a finite index-set F, satisfying

fi = (product of y_mdi) modulo (f1,...,f(i-1))
where the product runs over all m=1,...,n; and all d in F.

Procedures:

`nonMonomials(id)`
: non-monomials of the vanishing ideal id of a set of points

`cornerMonomials(N)`
: corner-monomials of the set of non-monomials N

`facGBIdeal(id)`
: GB G of id and linear factors of each element of G

D.4.21 primdec_lib

Library: primdec.lib

Purpose: Primary Decomposition and Radical of Ideals

Authors: Gerhard Pfister, pfister@mathematik.uni-kl.de (GTZ)
Wolfram Decker, decker@math.uni-sb.de (SY)
Hans Schoenemann, hannes@mathematik.uni-kl.de (SY)
Santiago Laplagne, slaplagn@dm.uba.ar (GTZ)

Overview: Algorithms for primary decomposition based on the ideas of Gianni, Trager and Zacharias (implementation by Gerhard Pfister), respectively based on the ideas of Shimoyama and Yokoyama (implementation by Wolfram Decker and Hans Schoenemann).
The procedures are implemented to be used in characteristic 0.
They also work in positive characteristic >> 0.
In small characteristic and for algebraic extensions, primdecGTZ may not terminate.
Algorithms for the computation of the radical based on the ideas of Krick, Logar, Laplagne and Kemper (implementation by Gerhard Pfister and Santiago Laplagne).
They work in any characteristic.
Baserings must have a global ordering and no quotient ideal. Exceptions: primdecGTZ, absPrimdecGTZ, minAssGTZ, primdecSY, minAssChar, radical accept non-global ordering.

Procedures:

`Ann(M)` annihilator of R^n/M, R=basering, M in R^n

`primdecGTZ(I)`
complete primary decomposition via Gianni,Trager,Zacharias

`primdecGTZE(I)`
complete primary decomposition via Gianni,Trager,Zacharias. Returns empty list for the unit ideal

`primdecSY(I...)`
complete primary decomposition via Shimoyama-Yokoyama

`primdecSYE(I,..)`
complete primary decomposition via Shimoyama-Yokoyama. Returns empty list for the unit ideal

`minAssGTZ(I)`
the minimal associated primes via Gianni,Trager,Zacharias (with modifications by Laplagne)

`minAssGTZE(I)`
the minimal associated primes via Gianni,Trager,Zacharias. Returns empty list for unit ideal

`minAssChar(I...)`
the minimal associated primes using characteristic sets

`minAssCharE(I..)`
the minimal associated primes using characteristic sets. Returns empty list for unit ideal

`testPrimary(L,k)`
tests the result of the primary decomposition

`testPrimaryE(L,k)`
tests the result of the primary decomposition. Handles also empty list L.

`radical(I)`
computes the radical of I via Krick/Logar (with modifications by Laplagne) and Kemper

`radicalEHV(I)`
computes the radical of I via Eisenbud,Huneke,Vasconcelos

`equiRadical(I)`
: the radical of the equidimensional part of the ideal I

`prepareAss(I)`
: list of radicals of the equidimensional components of I

`equidim(I)`
: weak equidimensional decomposition of I

`equidimMax(I)`
: equidimensional locus of I

`equidimMaxEHV(I)`
: equidimensional locus of I via Eisenbud,Huneke,Vasconcelos

`zerodec(I)`
: zerodimensional decomposition via Monico

`absPrimdecGTZ(I)`
: the absolute prime components of I

`absPrimdecGTZE(I)`
: the absolute prime components of I. Assumes I is not unit ideal.

`sep(f,k)` the separabel part of f as polynomial in Fp(t1,...,tm)

See also: Section D.4.22 [primdecint_lib], page 758.

D.4.22 primdecint_lib

Library: primdecint.lib

Purpose: primary decomposition of an ideal in the polynomial ring over the integers

Authors: G. Pfister pfister@mathematik.uni-kl.de
A. Sadiq afshanatiq@gmail.com
S. Steidel steidel@mathematik.uni-kl.de

Overview: A library for computing the primary decomposition of an ideal in the polynomial ring over the integers, Z[x_1,...,x_n].
The first procedure 'primdecZ' can be used in parallel.

Reference: Pfister, Sadiq, Steidel , "An Algorithm for primary decomposition in polynomial rings over the integers" , arXiv:1008.2074

Procedures:

`primdecZ(I)`
: compute the primary decomposition of ideal I

`primdecZM(I)`
: compute the primary decomposition of module I

`minAssZ(I)`
: compute the minimal associated primes of I

`radicalZ(I)`
: compute the radical of I

`heightZ(I)`
: compute the height of I

`equidimZ(I)`
compute the equidimensional part of I

`intersectZ(I,J)`
compute the intersection of I and J

See also: Section D.4.21 [primdec_lib], page 756.

D.4.23 primitiv_lib

Library: primitiv.lib

Purpose: Computing a Primitive Element

Author: Martin Lamm, email: lamm@mathematik.uni-kl.de

Procedures:

`primitive(ideal i)`
find minimal polynomial for a primitive element

`primitive_extra(i)`
find primitive element for two generators

`splitring(f,R[,L])`
define ring extension with name R and switch to it

D.4.24 realrad_lib

Library: realrad.lib

Purpose: Computation of real radicals

Author : Silke Spang

Overview: Algorithms about the computation of the real
radical of an arbitary ideal over the rational numbers and transcendetal extensions thereof

Procedures:

`realpoly(f)`
Computes the real part of the univariate polynomial f

`realzero(j)`
Computes the real radical of the zerodimensional ideal j

`realrad(j)`
Computes the real radical of an arbitary ideal over transcendental extension of the rational numbers

D.4.25 reesclos_lib

Library: reesclos.lib

Purpose: procedures to compute the int. closure of an ideal

Author: Tobias Hirsch, email: hirsch@math.tu-cottbus.de
Janko Boehm, email: boehm@mathematik.uni-kl.de
Magdaleen Marais, email: magdaleen@aims.ac.za

Overview: A library to compute the integral closure of an ideal I in a polynomial ring R=k[x(1),...,x(n)] using the Rees Algebra R[It] of I. It computes the integral closure of R[It],
which is a graded subalgebra of R[t]. The degree-k-component is the integral closure of the k-th power of I.

In contrast to the previous version, the library uses 'normal.lib' to compute the integral closure of R[It]. This improves the performance considerably.

Procedures:

`ReesAlgebra(I)`
: computes the Rees Algebra of an ideal I

`normalI(I[,p[,r]])`
: computes the integral closure of an ideal I using R[It]

D.4.26 sagbi_lib

Library: sagbi.lib

Purpose: Compute SAGBI basis (subalgebra bases analogous to Groebner bases for ideals) of a subalgebra

Authors: Jan Hackfeld, Jan.Hackfeld@rwth-aachen.de
Gerhard Pfister, pfister@mathematik.uni-kl.de
Viktor Levandovskyy, levandov@math.rwth-aachen.de

Overview: SAGBI stands for 'subalgebra bases analogous to Groebner bases for ideals'. SAGBI bases provide important tools for working with finitely presented subalgebras of a polynomial ring. Note, that in contrast to Groebner bases, SAGBI bases may be infinite.

References:
Ana Bravo: Some Facts About Canonical Subalgebra Bases, MSRI Publications 51, p. 247-254

Procedures:

`sagbiSPoly(A [,r,m])`
: computes SAGBI S-polynomials of A

`sagbiReduce(I,A [,t,mt])`
: performs subalgebra reduction of I by A

`sagbi(A [,m,t])`
: computes SAGBI basis for A

`sagbiPart(A,k[,m])`
: computes partial SAGBI basis for A

`algebraicDependence(I,it)`
: performs iterations of SAGBI for algebraic dependencies of I

See also: Section D.4.2 [algebra_lib], page 741.

D.4.27 sing4ti2_lib

Library: sing4ti2.lib

Purpose: Communication Interface to 4ti2

Authors: Thomas Kahle , kahle@mis.mpg.de
Anne Fruehbis-Krueger, anne@math.uni-hannover.de

Note: This library uses the external program 4ti2 for calculations and the standard unix tools sed and awk for conversion of the returned result

Procedures:

`markov4ti2(A[,i])`
compute Markov basis of given lattice

`hilbert4ti2(A[,i])`
compute Hilbert basis of given lattice

`graver4ti2(A[,i])`
compute Graver basis of given lattice

D.4.28 symodstd_lib

Library: symodstd.lib

Purpose: Procedures for computing Groebner basis of ideals being invariant under certain variable permutations.

Author: Stefan Steidel, steidel@mathematik.uni-kl.de

Overview: A library for computing the Groebner basis of an ideal in the polynomial ring over the rational numbers, that is invariant under certain permutations of the variables, using the symmetry and modular methods. More precisely let I = <f1,...,fr> be an ideal in Q[x(1),...,x(n)] and sigma a permutation of order k in Sym(n) such that sigma(I) = I. We assume that sigma({f1,...,fr}) = {f1,...,fr}. This can always be obtained by adding sigma(fi) to {f1,...,fr}.
To compute a standard basis of I we apply a modification of the modular version of the standard basis algorithm (improving the calculations in positive characteristic). Therefore we only allow primes p such that p-1 is divisible by k. This guarantees the existance of a k-th primitive root of unity in Z/pZ.

Procedures:

`genSymId(I,sigma)`
compute ideal J such that sigma(J) = J and J includes I

`isSymmetric(I,sigma)`
check if I is invariant under permutation sigma

`primRoot(p,k)`
int describing a k-th primitive root of unity in Z/pZ

`eigenvalues(I,sigma)`
list of eigenvalues of generators of I under sigma

`symmStd(I,sigma)`
standard basis of I using invariance of I under sigma

`syModStd(I,sigma)`
SB of I using modular methods and sigma(I) = I

D.4.29 toric_lib

Library: toric.lib

Purpose: Standard Basis of Toric Ideals

Author: Christine Theis, email: ctheis@math.uni-sb.de

Procedures:

`toric_ideal(A,..)`
computes the toric ideal of A

`toric_std(ideal I)`
standard basis of I by a specialized Buchberger algorithm

D.5 Algebraic geometry

D.5.1 numerAlg_lib

Todos/Issues:

does not follow the naming convention

syntax errors in examples

no test suite

Library: NumerAlg.lib

Purpose: Numerical Algebraic Algorithm

Overview: The library contains procedures to
test the inclusion, the equality of two ideals defined by polynomial systems, compute the degree of a pure i-dimensional component of an algebraic variety defined by a polynomial system,
compute the local dimension of an algebraic variety defined by a polynomial system at a point computed as an approximate value. The use of the library requires to install Bertini (http://www.nd.edu/~sommese/bertini).

Author: Shawki AlRashed, rashed@mathematik.uni-kl.de; sh.shawki@yahoo.de

Procedures:

`Incl(ideal I, ideal J)`
test if I containes J

`Equal(ideal I, ideal J)`
test if I equals to J

`DegreePure(ideal I, int i)`
computes the degree of a pure i-dimensional

`NumLocalDim(ideal I, p)`
numerical local dimension at a point computed as an approximate value

D.5.2 numerDecom_lib

Todos/Issues:

- does not follow the naming convention
- syntax errors in examples
- no test suite

Library: NumDecom.lib

Purpose: Numerical Decomposition of Ideals

Overview: The library contains procedures to compute
numerical irreducible decomposition, and
numerical primary decomposition of an algebraic variety defined by a polynomial system. The use of the library requires to install Bertini (@uref{http://www.nd.edu/~sommese/bertini}).

Author: Shawki AlRashed, rashed@mathematik.uni-kl.de; sh.shawki@yahoo.de

Procedures:

`re2squ(ideal I)`
: reduction to square system

`UseBertini(ideal H,string sv)`
: use Bertini to compute the solutions of the homotopy function

`Singular2bertini(list L)`
: adopt the list to be a read file in Bertini as a start solution set

`bertini2Singular(string snp, int q)`
: adopt the file of solutions of the homotopy function to be a list in SINGULAR

`ReJunkUseHomo(ideal I, ideal L, list W, list w)`
: remove junk points using the homotopy function

`JuReTopDim(ideal J,list w,int tt, int d)`
: remove junk points that are on top-dimensional component

`JuReZeroDim(ideal J,list w, int d)`
: remove junk points from 0-dimensional component

`WitSupSet(ideal I)`
: witness point super set

`WitSet(ideal I)`
: witness point set

`NumIrrDecom(ideal I)`
: numerical irreducible decomposition

`defl(ideal I, int d)`
: deflation of ideal I

`NumPrimDecom(ideal I, int d)`
: numerical primary decomposition

D.5.3 orbitparam_lib

Library: orbitparam.lib

Purpose: Parametrizing orbits of unipotent actions

Authors: J. Boehm, boehm at mathematik.uni-kl.de
S. Papadakis, papadak at math.ist.utl.pt

Overview: This library implements the theorem of Chevalley-Rosenlicht as stated in Theorem 3.1.4 of [Corwin, Greenleaf]. Given a set of strictly upper triangular n x n matrices L_1,...,L_c which generate a Lie algebra as a vector space, and a vector v of size n, the function `parametrizeOrbit` constructs a parametrization of the orbit of v under the action of exp(<L_1,...,L_c>).

To compute exp of the Lie algebra elements corresponding to the parameters we require that the characteristic of the base field is zero or larger than n.

By determining the parameters from bottom to top
this allows you to find an element in the orbit with (at least) as many zeros as the dimension of the orbit.

Note: Theorem 3.1.4 of [Corwin, Greenleaf] uses strictly lower triangular matrices.

References:

Laurence Corwin, Frederick P. Greenleaf: Representations of Nilpotent Lie Groups and their Applications: Volume 1, Part 1, Basic Theory and Examples, Cambridge University Press (2004).

Procedures:

`tangentGens(list,matrix)`
Returns elements in the Lie algebra, which form a basis of the tangent space of the parametrization.

`matrixExp(matrix)`
Matrix exp for nilpotent matrices.

`matrixLog(matrix)`
Matrix log for unipotent matrices.

`parametrizeOrbit(list,matrix)`
Returns parametrization of the orbit.

`maxZeros(list,matrix)`
Determine an element in the orbit with the maximum number of zeroes.

D.5.4 paraplanecurves_lib

Library: paraplanecurves.lib

Purpose: Rational parametrization of rational plane curves

Authors: J. Boehm, boehm at mathematik.uni-kl.de
W. Decker, decker at mathematik.uni-kl.de
S. Laplagne, slaplagn at dm.uba.ar
F. Seelisch, seelisch at mathematik.uni-kl.de

Overview: Suppose C = {f(x,y,z)=0} is a rational plane curve, where f is homogeneous of degree n with coefficients in Q and absolutely irreducible (these conditions are checked automatically.)
After a first step, realized by a projective automorphism in the procedure adjointIdeal, C satisfies:
- C does not have singularities at infinity z=0.
- C does not contain the point (0:1:0) (that is, the dehomogenization of f with respect to z is monic as a polynomial in y).
Considering C in the chart z<>0, the algorithm regards x as transcendental and y as algebraic and computes an integral basis in C(x)[y] of the integral closure of C[x] in C(x,y) using the normalization algorithm from Section D.4.18 [normal_lib], page 753: see Section D.4.9 [integralbasis_lib], page 747. In a future edition of the library, also van Hoeij's algorithm for computing the integral basis will be available.
From the integral basis, the adjoint ideal is obtained by linear algebra. Alternatively, the algorithm starts with a local analysis of the singular locus of C. Then, for each primary component of the singular locus which does not correspond to ordinary multiple points or cusps, the integral basis algorithm is applied separately. The ordinary multiple points and cusps, in turn, are addressed by a straightforward direct algorithm. The adjoint ideal is obtained by intersecting all ideals obtained locally. The local variant of the algorithm is used by default.
The linear system corresponding to the adjoint ideal maps the curve birationally to a rational normal curve in P^(n-2).
Iterating the anticanonical map, the algorithm projects the rational normal curve to PP1 for n odd resp. to a conic C2 in PP2 for n even.
In case n is even, the algorithm tests whether there is a rational point on C2 and if so gives a parametrization of C2 which is defined over Q. Otherwise, the parametrization given is defined over a quadratic field extension of Q.
By inverting the birational map of C to PP1 resp. to C2, a parametrization of C is obtained (defined over Q or the quadratic field extension).

References:

Janko Boehm: Parametrisierung rationaler Kurven, Diploma Thesis, http://www.math.uni-sb.de/ag/schreyer/jb/diplom%20janko%20boehm.pdf

Janko Boehm, Wolfram Decker, Santiago Laplagne, Gerhard Pfister: Local to global algorithms for the Gorenstein adjoint ideal of a curve, Algorithmic and Experimental Methods in Algebra, Geometry, and Number Theory, Springer 2018

Theo de Jong: An algorithm for computing the integral closure, Journal of Symbolic Computation 26 (3) (1998), p. 273-277

Gert-Martin Greuel, Santiago Laplagne, Frank Seelisch: Normalization of Rings, Journal of Symbolic Computation 9 (2010), p. 887-901

Mark van Hoeij: An Algorithm for Computing an Integral Basis in an Algebraic Function Field, Journal of Symbolic Computation 18 (1994), p. 353-363, http://www.math.fsu.edu/~hoeij/papers/comments/jsc1994.html

Procedures:

`adjointIdeal(poly, [...])`
Adjoint ideal of a plane curve

`invertBirMap(ideal,ideal)`
Invert a birational map of algebraic varieties

`paraPlaneCurve(poly, [...])`

Compute a rational parametrization of a rational plane curve

`rncAntiCanonicalMap(ideal)`
Anticanonical map of a rational normal curve

`rationalPointConic(poly)`
Finds a point on the conic. This point has either coefficients in Q or in a quadratic extension field of Q

`mapToRatNormCurve(poly,ideal)`
Map a plane rational curve to a rational normal curve (RNC)

`rncItProjOdd(ideal)`
Map a RNC via successive anticanonical maps to PP1

`rncItProjEven(ideal)`
Map a RNC via successive anticanonical maps to a conic in PP2

`paraConic(poly)`
Compute a rational parametrization of a conic

`testParametrization(poly,ring)`
Checks whether a given curve is parametrized by a given rational map (defined in the given ring)

`testPointConic(poly,ring)`
Checks whether a given point (defined in the given ring) lies on the given conic.

D.5.5 resbinomial_lib

Todos/Issues:

formatting is inapproprociate

avoid export

bad names(or should be static): identifyvars, elimrep, convertdata, lcmofall, genoutput, salida, iniD, reslist, sumlist, dividelist, createlist

Library: resbinomial.lib

Purpose: Combinatorial algorithm of resolution of singularities of binomial ideals in arbitrary characteristic. Binomial resolution algorithm of Blanco

Authors: R. Blanco, mariarocio.blanco@uclm.es,
G. Pfister, pfister@mathematik.uni-kl.de

Procedures:

`BINresol(J)`
computes a E-resolution of singularities of (J) (THE SECOND PART IS NOT IMPLEMENTED YET)

`Eresol(J)`
computes a E-resolution of singularities of (J) in char 0

`determinecenter(L1,L2,c,n,Y,a,mb,flag,control3)`
computes the next blowing-up center

`Blowupcenter(L1,id,m,L2,c,n,h)`
makes the blowing-up

`Nonhyp(Coef,expJ,sJ,n,flag,sums)`
computes the ideal generated by the non hyperbolic generators of expJ

`identifyvar()`
identifies status of variables

`Edatalist(Coef,Exp,k,n,flag)`
gives the E-order of each term in Exp

`EOrdlist(Coef,Exp,k,n,flag)`
computes the E-order of an ideal (giving in the language of lists)

`maxEord(Coef,Exp,k,n,flag)`
computes de maximum E-order of an ideal given by Coef and Exp

`ECoef(Coef,expP,sP,V,auxc,n,flag)`
Computes a simplified version of the E-Coeff ideal. The E-orders are correct, but tranformations of coefficients of the generators and powers of binomials cannot be computed easily in terms of lists.

`elimrep(L)`
removes repeated terms from a list

`Emaxcont(Coef,Exp,k,n,flag)`
computes a list of hypersurfaces of E-maximal contact

`cleanunit(mon,n,flag)`
clean the units in a monomial mon

`resfunction(t,auxinv,nchart,n)`
composes the E-resolution function

`calculateI(Coef,J,c,n,Y,a,b,D)`
computes the order of the non monomial part of an ideal J

`Maxord(L,n)`
computes the maximum exponent of an exceptional monomial ideal

`Gamma(L,c,n)`
computes the Gamma function for an exceptional monomial ideal given by L

`convertdata(C,L,n,flag)`
computes the ideal corresponding to C,L

`lcmofall(nchart,mobile)`
computes the lcm of the denominators of the E-orders for all the charts

`computemcm(Eolist)`
computes the lcm of the denominators of the E-orders for one chart

`constructH(Hhist,n,flag)`
construct the list of exceptional divisors accumulated at this chart

`constructblwup(blwhist,n,chy,flag)`
construct the ideal defining the map K[W] –> K[Wi], which gives the composition map of all the blowing up leading to this chart

`constructlastblwup(blwhist,n,chy,flag)`
construct the ideal defining the last blowup leading to this chart

`genoutput(chart,mobile,nchart,nsons,n,q,p)`
: generates the output for visualization

`salida(idchart,chart,mobile,numson,previousa,n,q)`
: generates the output for one chart

`iniD(n)` creates a list of lists of zeros of size n

`sumlist(L1,L2)`
: sums two lists component to component

`reslist(L1,L2)`
: subtracts two lists component to component

`multiplylist(L,a)`
: multiplies a list by a number, component to component

`dividelist(L1,L2)`
: divides two lists component to component

`createlist(L1,L2)`
: creates a list of lists of two elements

D.5.6 resgraph_lib

Library: resgraph.lib

Purpose: Visualization of Resolution Data

Author: A. Fruehbis-Krueger, anne@mathematik.uni-kl.de,

Note: This library uses the external programs surf, graphviz and imagemagick.
Input data is assumed to originate from resolve.lib and reszeta.lib

Procedures:

`InterDiv(M[,name])`
: dual graph of resolution of a surface (uses graphviz,imagemagick)

`ResTree(L,M[,name])`
: tree of charts of resolution (uses graphviz,imagemagick)

`finalCharts(L,...)`
: pictures of final charts of surface (uses surf)

D.5.7 resjung_lib

Library: jung.lib

Purpose: Resolution of surface singularities (Desingularization) Algorithm of Jung

Author: Philipp Renner, philipp_renner@web.de

Procedures:

`jungresolve(J[,is_noeth])`
: computes a resolution (!not a strong one) of the surface given by the ideal J using Jungs Method,

`jungnormal(J[,is_noeth])`
: computes a representation of J such that all it's singularities are of Hirzebruch-Jung type,

`jungfib(J[,is_noeth])`
: computes a representation of J such that all it's singularities are quasi-ordinary

D.5.8 resolve_lib

Library: resolve.lib

Purpose: Resolution of singularities (Desingularization) Algorithm of Villamayor

Authors: A. Fruehbis-Krueger, anne@mathematik.uni-kl.de,
G. Pfister, pfister@mathematik.uni-kl.de

References:

[1] J.Kollar: Lectures on Resolution of Singularities, Princeton University Press (2007)
(contains large overview over various known methods for curves and surfaces as well as a detailed description of the approach in the general case)
[2] A.Bravo, S.Encinas, O.Villamayor: A Simplified Proof of Desingularisation and Applications, Rev. Math. Iberoamericana 21 (2005), 349-458
(description of the algorithmic proof of desingularization in characteristic zero which underlies this implementation)
[3] A.Fruehbis-Krueger: Computational Aspects of Singularities, in J.-P. Brasselet, J.Damon et al.: Singularities in Geometry and Topology, World Scientific Publishing, 253–327 (2007)
(chapter 4 contains a detailed discussion on algorithmic desingularization and efficiency aspects thereof)

Procedures:

`blowUp(J,C[,W,E])`
: computes the blowing up of the variety V(J) (considered as embedded in V(W)) in the (smooth) center V(C),

`blowUp2(J,C)`
: computes the blowing up of the variety V(J) in the (possibly singular) center V(C)

`Center(J[,W,E])`
: computes 'Villamayor'-center for blow up

`resolve(J)`
: computes the desingularization of the variety V(J)

`showBO(BO)`
: prints the content of a BO in more human readable form

`presentTree(L)`
: prints the final charts in more human readable form

`showDataTypes()`
: prints help text for output data types

`blowUpBO(BO,C)`
: computes the blowing up of the variety V(BO[1]) in the center V(C). BO is a list (basic object), C is an ideal

`createBO(J,W,E)`
: creates basic object from input data

`CenterBO(BO)`
: computes the center for the next blow-up of the given basic object

`Delta(BO)`
: apply the Delta-operator of [Bravo,Encinas,Villamayor]

`DeltaList(BO)`
: list of results of Delta^0 to Delta^bmax

D.5.9 reszeta_lib

Library: reszeta.lib

Purpose: topological Zeta-function and some other applications of desingularization

Authors: A. Fruehbis-Krueger, anne@mathematik.uni-kl.de,
G. Pfister, pfister@mathematik.uni-kl.de

References:

[1] Fruehbis-Krueger,A., Pfister,G.: Some Applications of Resolution of Singularities from a Practical Point of View, in Computational Commutative and Non-commutative Algebraic Geometry, NATO Science Series III, Computer and Systems Sciences 196, 104-117 (2005) [2] Fruehbis-Krueger: An Application of Resolution of Singularities: Computing the topological Zeta-function of isolated surface singularities in (C^3,0), in D.Cheniot, N.Dutertre et al.(Editors): Singularity Theory, World Scientific Publishing (2007)

Procedures:

`intersectionDiv(L)`
: computes intersection form and genera of exceptional divisors (isolated singularities of surfaces)

`spectralNeg(L)`
: computes negative spectral numbers (isolated hypersurface singularity)

`discrepancy(L)`
: computes discrepancy of given resolution

`zetaDL(L,d)`
: computes Denef-Loeser zeta function (hypersurface singularity of dimension 2)

`collectDiv(L[,iv])`
: identify exceptional divisors in different charts (embedded and non-embedded case)

`prepEmbDiv(L[,b])`
: prepare list of divisors (including components of strict transform, embedded case)

`abstractR(L)`
: pass from embedded to non-embedded resolution

`computeV(re,DL)`
: multiplicities of divisors in pullback of volume form

`computeN(re,DL)`
: multiplicities of divisors in total transform of resolution

D.5.10 sheafcoh_lib

Library: sheafcoh.lib

Purpose: Procedures for Computing Sheaf Cohomology

Authors: Wolfram Decker, decker@mathematik.uni-kl.de
Christoph Lossen, lossen@mathematik.uni-kl.de
Gerhard Pfister, pfister@mathematik.uni-kl.de
Oleksandr Motsak, U@D, where U={motsak}, D={mathematik.uni-kl.de}

Procedures:

`truncate(phi,d)`
truncation of coker(phi) at d

`truncateFast(phi,d)`
truncation of coker(phi) at d (fast+ experimental)

`CM_regularity(M)`
Castelnuovo-Mumford regularity of coker(M)

`sheafCohBGG(M,l,h)`
cohomology of sheaf associated to coker(M)

`sheafCohBGG2(M,l,h)`
cohomology of sheaf associated to coker(M), experimental version

`sheafCoh(M,l,h)`
cohomology of sheaf associated to coker(M)

`dimH(i,M,d)`
compute h^i(F(d)), F sheaf associated to coker(M)

`dimGradedPart()`

`displayCohom(B,l,h,n)`
display intmat as Betti diagram (with zero rows)

D.5.11 JMBTest_lib

Library: JMBTest.lib

Purpose: A library for Singular which performs JM basis test.

Author: Michela Ceria, email: michela.ceria@unito.it

Overview: The library performs the J-marked basis test, as described in [CR], [BCLR]. Such a test is performed via the criterion explained in [BCLR], concerning Eliahou-Kervaire polynomials (EK from now on). We point out that all the polynomials are homogeneous and they must be arranged by degree.
The fundamental steps are the following:
-construct the Vm polynomials, via the algorithm VConstructor explained in [CR];
-construct the Eliahou-Kervaire polynomials defined in [BCLR];
-reduce the Eliahou-Kervaire polynomials using the Vm's;
-if it exist an Eliahou-Kervaire polynomial such that its reduction mod Vm is different from zero, the given one is not a J-Marked basis.

The algorithm terminates only if the ordering is rp. Anyway, the number of reduction steps is bounded.

References:

[CR] Francesca Cioffi, Margherita Roggero,Flat Families by Strongly Stable Ideals and a Generalization of Groebner Bases, J. Symbolic Comput. 46, 1070-1084, (2011).
[BCLR] Cristina Bertone, Francesca Cioffi, Paolo Lella, Margherita Roggero, Upgraded methods for the effective computation of marked schemes on a strongly stable ideal, Journal of Symbolic Computation
(2012), http://dx.doi.org/10.1016/j.jsc.2012.07.006

Procedures:

`Minimus(ideal)`
: minimal variable in an ideal

`Maximus(ideal)`
: maximal variable in an ideal

`StartOrderingV(list,list)`
: ordering of polynomials as in [BCLR]

`TestJMark(list)`
: tests whether we have a J-marked basis

See also: Section D.5.12 [JMSConst_lib], page 772.

D.5.12 JMSConst_lib

Library: JMSConst.lib

Purpose: A library for Singular which constructs J-Marked Schemes.

Author: Michela Ceria, email: michela.ceria@unito.it

Overview: The library performs the J-marked computation, as described in [BCLR]. As in JMBTest.lib we construct the V polynomials and we reduce the EK polynomials w.r.t. them, putting the coefficients as results.

The algorithm terminates only if the ordering is rp. Anyway, the number of reduction steps is bounded.

References:

[CR] Francesca Cioffi, Margherita Roggero,Flat Families by Strongly Stable Ideals and a Generalization of Groebner Bases, J. Symbolic Comput. 46, 1070-1084, (2011).
[BCLR] Cristina Bertone, Francesca Cioffi, Paolo Lella, Margherita Roggero, Upgraded methods for the effective computation of marked schemes on a strongly stable ideal, Journal of Symbolic Computation
(2012), http://dx.doi.org/10.1016/j.jsc.2012.07.006

Procedures:

`BorelCheck(ideal,r)`
: checks whether the given ideal is Borel

`JMarkedScheme(list, list, list, int)`
: computes authomatically all the J-marked scheme

See also: Section D.5.11 [JMBTest_lib], page 771; Section D.5.12 [JMSConst_lib], page 772.

D.6 Singularities

D.6.1 alexpoly_lib

Library: alexpoly.lib

Purpose: Resolution Graph and Alexander Polynomial

Author: Fernando Hernando Carrillo, hernando@agt.uva.es
Thomas Keilen, keilen@mathematik.uni-kl.de

Overview: A library for computing the resolution graph of a plane curve singularity f, the total multiplicities of the total transforms of the branches of f along the exceptional divisors of a minimal good resolution of f, the Alexander polynomial of f, and the zeta function of its monodromy operator.

Procedures:

`resolutiongraph(f)`
resolution graph f

`totalmultiplicities(f)`
resolution graph, total multiplicities and strict multiplicities of f

`alexanderpolynomial(f)`
Alexander polynomial of f

`semigroup(f)`
calculates generators for the semigroup of f

`proximitymatrix(f)`
calculates the proximity matrix of f

`multseq2charexp(v)`
converts multiplicity sequence to characteristic exponents

`charexp2multseq(v)`
converts characteristic exponents to multiplicity sequence

`charexp2generators(v)`
converts characteristic exponents to generators of the semigroup

`charexp2inter(c,e)`
converts contact matrix and charact. exp. to intersection matrix

`charexp2conductor(v)`
converts characteristic exponents to conductor

`charexp2poly(v,a)`
calculates a polynomial f with characteristic exponents v

`tau_es2(f)`
equisingular Tjurina number of f

D.6.2 arcpoint_lib

Library: arcpoint.lib

Purpose: Truncations of arcs at a singular point

Author: Nadine Cremer cremer@mathematik.uni-kl.de

Overview: An arc is given by a power series in one variable, say t, and truncating it at a positive integer i means cutting the t-powers `>` i. The set of arcs truncated at order `<bound>` is denoted Tr(i). An algorithm for computing these sets (which happen to be constructible) is given in [Lejeune-Jalabert, M.: Courbes trac'ees sur un germe d'hypersurface, American Journal of Mathematics, 112 (1990)]. Our procedures for computing the locally closed sets contributing to the set of truncations rely on this algorithm.

Procedures:

`nashmult(f,bound)`
: determines locally closed sets relevant for computing truncations of arcs over a hypersurface with isolated singularity defined by f. The sets are given by two ideals specifying relations between coefficients of power series in t. One of the ideals defines an open set, the other one the complement of a closed set within the open one. We consider only coefficients up to t^`<bound>`. Moreover, the sequence of Nash Multiplicities of each set is displayed

`removepower(I)`
: modifies the ideal I such that the algebraic set defined by it remains the same: removes powers of variables

`idealsimplify(I,maxiter)`
: further simplification of I in the above sense: reduction with other elements of I. The positive integer `<maxiter>` gives a bound to the number of repetition steps

`equalJinI(I,J)`
: tests if two ideals I and J are equal under the assumption that J is contained in I. Returns 1 if this is true and 0 otherwise

D.6.3 classify_lib

Library: classify.lib

Purpose: Arnold Classifier of Singularities

Author: Kai Krueger, krueger@mathematik.uni-kl.de

Overview: A library for classifying isolated hypersurface singularities w.r.t. right equivalence, based on the determinator of singularities by V.I. Arnold.

Procedures:

`basicinvariants(f)`
: computes Milnor number, determinacy-bound and corank of

`classify(f)`
: normal form of polynomial f determined with Arnold's method

`corank(f)`
: computes the corank of f (i.e. of the Hessian of f)

`Hcode(v)` coding of intvec v acoording to the number repetitions

`init_debug([n])`
: print trace and debugging information depending on int n

`internalfunctions()`
display names of internal procedures of this library

`milnorcode(f[,e])`
Hilbert polynomial of [e-th] Milnor algebra coded with Hcode

`morsesplit(f)`
residual part of f after applying the splitting lemma

`quickclass(f)`
normal form of f determined by invariants (milnorcode)

`singularity(s,[])`
normal form of singularity given by its name s and index

`A_L(s/f)` shortcut for quickclass(f) or normalform(s)

`normalform(s)`
normal form of singularity given by its name s

`debug_log(lev,[])`
print trace and debugging information w.r.t level>@DeBug

`swap(a,b)`
swaps the arguments

`modality(f)`
modality of the singularity

`complexSingType(f)`
complex type of the singularity as a string

`prepRealclassify(f)`
the modality and the complex type of the singularity at once

See also: Section D.15.31 [realclassify_lib], page 861.

D.6.4 classifyceq_lib

Library: classifyCeq.lib

Purpose: simple hypersurface singularities in characteristic p > 0

Authors: Deeba Afzal deebafzal@gmail.com
Faira Kanwal Janjua fairakanwaljanjua@gmail.com

Overview: A library for classifying the simple singularities with respect to contact equivalence in charateristic p > 0 . Simple hypersurface singularities in charateristic p > 0 were classified by Greuel and Kroening [1] with respect to contact equivalence. The classifier we use has been proposed in [2].

References:

[1] Greuel, G.-M.; Kroening, H.: Simple singularities in positive characteristic. Math.Z. 203, 339-354 (1990).
[2] Afzal,D.;Binyamin,M.A.;Janjua,F.K.: On the classification of simple singularities in positive characteristic.

Procedures:

`classifyCeq(f)`
simple hypersurface singularities in charateristic p > 0

D.6.5 classifyci_lib

Library: classifyci.lib

Purpose: Isolated complete intersection singularities in characteristic 0

Authors: Gerhard Pfister pfister@mathematik.uni-kl.de
Deeba Afzal deebafzal@gmail.com

Overview: A library for classifying isolated complete intersection singularities for the base field of characteristic 0 and for computing weierstrass semigroup of the space curve.Isolated complete intersection singularities were classified by M.Giusti [1] for the base field of characteristic 0. Algorithm for the semigroup of a space curve singularity is given in [2].

References:

[1] Giusti,M:Classification des singularities isolees simples d'intersections completes, C,R.Acad.Sci.Paris Ser.A-B 284(1977),167-169.
[2] Castellanos,A.,Castellanos,J.,2005:Algorithm for the semigroup of a space curve singularity. Semigroup Forum 70,44-66.

Procedures:

`classifyicis(I)`
Isolated simple complete intersection singularities for the base field of charateristic 0

`Semigroup(I)`
Weierstrass semigroup of the space curve given by equations

D.6.6 curvepar_lib

Library: curvepar.lib

Purpose: Resolution of space curve singularities, semi-group

Author: Gerhard Pfister email: pfister@mathematik.uni-kl.de Nil Sahin email: e150916@metu.edu.tr
Maryna Viazovska email: viazovsk@mathematik.uni-kl.de

Procedures:

`BlowingUp(f,I,l)`
BlowingUp of V(I) at the point 0;

`CurveRes(I)`
Resolution of V(I)

`CurveParam(I)`
Parametrization of algebraic branches of V(I)

`WSemigroup(X,b)`
Weierstrass semigroup of the curve

`primparam(x,y,c)`
HN matrix of parametrization(x(t),y(t))

`MultiplicitySequence(I)`
Multiplicity sequences of the branches of plane curve V(I)

`CharacteristicExponents(I)`
: Characteristic exponents of the branches of plane curve V(I)

`IntersectionMatrix(I)`
: Intersection Matrix of the branches of plane curve V(I)

`ContactMatrix(I)`
: Contact Matrix of the branches of plane curve V(I)

`plainInvariants(I)`
: Invariants of the branches of plane curve V(I)

See also: Section D.6.16 [spcurve_lib], page 784.

D.6.7 deform_lib

Library: deform.lib

Purpose: Miniversal Deformation of Singularities and Modules

Author: Bernd Martin, email: martin@math.tu-cottbus.de

Procedures:

`versal(Fo[,d,any])`
: miniversal deformation of isolated singularity Fo

`mod_versal(Mo,I,[,d,any])`
: miniversal deformation of module Mo modulo ideal I

`lift_kbase(N,M)`
: lifting N into standard kbase of M

`lift_rel_kb(N,M[,kbM,p])`
: relative lifting N into a kbase of M

D.6.8 equising_lib

Library: equising.lib

Purpose: Equisingularity Stratum of a Family of Plane Curves

Author: Christoph Lossen, lossen@mathematik.uni-kl.de
Andrea Mindnich, mindnich@mathematik.uni-kl.de

Procedures:

`tau_es(f)`
: codim of mu-const stratum in semi-universal def. base

`esIdeal(f)`
: (Wahl's) equisingularity ideal of f

`esStratum(F[,m,L])`
: equisingularity stratum of a family F

`isEquising(F[,m,L])`
: tests if a given deformation is equisingular

`control_Matrix(M)`
: computes list of blowing-up data

D.6.9 gmssing_lib

Library: gmssing.lib

Purpose: Gauss-Manin System of Isolated Singularities

Author: Mathias Schulze, mschulze at mathematik.uni-kl.de

Overview: A library for computing invariants related to the Gauss-Manin system of an isolated hypersurface singularity.

References:

[Sch01] M. Schulze: Algorithms for the Gauss-Manin connection. J. Symb. Comp. 32,5 (2001), 549-564.
[Sch02] M. Schulze: The differential structure of the Brieskorn lattice. In: A.M. Cohen et al.: Mathematical Software - ICMS 2002. World Scientific (2002).
[Sch03] M. Schulze: Monodromy of Hypersurface Singularities. Acta Appl. Math. 75 (2003), 3-13.
[Sch04] M. Schulze: A normal form algorithm for the Brieskorn lattice. J. Symb. Comp. 38,4 (2004), 1207-1225.

Procedures:

`gmsring(t,s)`
Gauss-Manin system of t with variable s

`gmsnf(p,K)`
Gauss-Manin normal form of p

`gmscoeffs(p,K)`
Gauss-Manin basis representation of p

`bernstein(t)`
Bernstein-Sato polynomial of t

`monodromy(t)`
Jordan data of complex monodromy of t

`spectrum(t)`
singularity spectrum of t

`sppairs(t)`
spectral pairs of t

`vfilt(t)` V-filtration of t on Brieskorn lattice

`vwfilt(t)`
weighted V-filtration of t on Brieskorn lattice

`tmatrix(t)`
matrix of t w.r.t. good basis of Brieskorn lattice

`endvfilt(V)`
endomorphism V-filtration on Jacobian algebra

`sppnf(a,w[,m])`
spectral pairs normal form of (a,w[,m])

`sppprint(spp)`
print spectral pairs spp

`spadd(sp1,sp2)`
: sum of spectra sp1 and sp2

`spsub(sp1,sp2)`
: difference of spectra sp1 and sp2

`spmul(sp0,k)`
: linear combination of spectra sp

`spissemicont(sp[,opt])`
: semicontinuity test of spectrum sp

`spsemicont(sp0,sp[,opt])`
: semicontinuous combinations of spectra sp0 in sp

`spmilnor(sp)`
: Milnor number of spectrum sp

`spgeomgenus(sp)`
: geometrical genus of spectrum sp

`spgamma(sp)`
: gamma invariant of spectrum sp

See also: Section 7.7.4 [dmod_lib], page 400; Section D.6.10 [gmspoly_lib], page 779; Section D.6.13 [mondromy_lib], page 782; Section D.6.17 [spectrum_lib], page 784.

D.6.10 gmspoly_lib

Library: gmspoly.lib

Purpose: Gauss-Manin System of Tame Polynomials

Author: Mathias Schulze, mschulze at mathematik.uni-kl.de

Overview: A library for computing the Gauss-Manin system of a cohomologically tame polynomial f. Schulze's algorithm [Sch05], based on Sabbah's theory [Sab98], is used to compute a good basis of (the Brieskorn lattice of) the Gauss-Manin system and the differential operation of f in terms of this basis. In addition, there is a test for tameness in the sense of Broughton. Tame polynomials can be considered as an affine algebraic analogue of local analytic isolated hypersurface singularities. They have only finitely many citical points, and those at infinity do not give rise to atypical values in a sense depending on the precise notion of tameness considered. Well-known notions of tameness like tameness, M-tameness, Malgrange-tameness, and cohomological tameness, and their relations, are reviewed in [Sab98,8]. For ordinary tameness, see Broughton [Bro88,3]. Sabbah [Sab98] showed that the Gauss-Manin system, the D-module direct image of the structure sheaf, of a cohomologically tame polynomial carries a similar structure as in the isolated singularity case, coming from a Mixed Hodge structure on the cohomology of the Milnor (typical) fibre (see gmssing.lib). The data computed by this library encodes the differential structure of the Gauss-Manin system, and the Mixed Hodge structure of the Milnor fibre over the complex numbers. As a consequence, it yields the Hodge numbers, spectral pairs, and monodromy at infinity.

References:

[Bro88] S. Broughton: Milnor numbers and the topology of polynomial hypersurfaces. Inv. Math. 92 (1988) 217-241.

[Sab98] C. Sabbah: Hypergeometric periods for a tame polynomial. arXiv.org

math.AG/9805077.
[Sch05] M. Schulze: Good bases for tame polynomials. J. Symb. Comp. 39,1 (2005), 103-126.

Procedures:

`isTame(f)`
: test whether the polynomial f is tame

`goodBasis(f)`
: good basis of Brieskorn lattice of cohom. tame polynomial f

See also: Section D.6.9 [gmssing_lib], page 778.

D.6.11 hnoether_lib

Library: hnoether.lib

Purpose: Hamburger-Noether (Puiseux) Expansion

Authors: Martin Lamm, lamm@mathematik.uni-kl.de
Christoph Lossen, lossen@mathematik.uni-kl.de

Overview: A library for computing the Hamburger-Noether expansion (analogue of Puiseux expansion over fields of arbitrary characteristic) of a reduced plane curve singularity following [Campillo, A.: Algebroid curves in positive characteristic, Springer LNM 813 (1980)].
The library contains also procedures for computing the (topological) numerical invariants of plane curve singularities.

Procedures:

`hnexpansion(f [,"ess"])`
: Hamburger-Noether (HN) expansion of f

`develop(f [,n])`
: HN expansion of irreducible plane curve germs

`extdevelop(hne,n)`
: extension of the H-N expansion hne of f

`param(hne [,s])`
: parametrization of branches described by HN data

`displayHNE(hne)`
: display HN expansion as an ideal

`invariants(hne)`
: invariants of f, e.g. the characteristic exponents

`displayInvariants(hne)`
: display invariants of f

`multsequence(hne)`
: sequence of multiplicities

`displayMultsequence(hne)`
: display sequence of multiplicities

`intersection(hne1,hne2)`
: intersection multiplicity of two local branches

`is_irred(f)`
: test whether f is irreducible as power series

`delta(f)`
: delta invariant of f

`newtonpoly(f)`
: (local) Newton polygon of f

`is_NND(f)`
: test whether f is Newton non-degenerate

`stripHNE(hne)`
: reduce amount of memory consumed by hne

`puiseux2generators(m,n)`
: convert Puiseux pairs to generators of semigroup

`separateHNE(hne1,hne2)`
: number of quadratic transf. needed for separation

`squarefree(f)`
: a squarefree divisor of the polynomial f

`allsquarefree(f,l)`
: the maximal squarefree divisor of the polynomial f

`further_hn_proc()`
: show further procedures useful for interactive use

D.6.12 kskernel_lib

Library: kskernel.lib

Purpose: procedures for computing the kernel of the kodaira-spencer map

Author: Tetyana Povalyaeva, povalyae@mathematik.uni-kl.de

Procedures:

`KSker(p,q)`
: kernel of the Kodaira-Spencer map of a versal deformation of an irreducible plane curve singularity

`KSconvert(M)`
: kernel of the Kodaira-Spencer map in quasihomogeneous variables T with corresponding negative degrees

`KSlinear(M)`
: matrix of linear terms of the kernel of the Kodaira-Spencer map

`KScoef(i,j,P,Q,qq)`
: coefficient of the given term in the matrix of kernel of the Kodaira-Spencer map

`StringF(i,j,p,q)`
: expression in variables T(i) with non-resolved brackets for the further computation of coefficient in the matrix of kernel of the Kodaira-Spencer map

D.6.13 mondromy_lib

Library: mondromy.lib

Purpose: Monodromy of an Isolated Hypersurface Singularity

Author: Mathias Schulze, email: mschulze@mathematik.uni-kl.de

Overview: A library to compute the monodromy of an isolated hypersurface singularity. It uses an algorithm by Brieskorn (manuscripta math. 2 (1970), 103-161) to compute a connection matrix of the meromorphic Gauss-Manin connection up to arbitrarily high order, and an algorithm of Gerard and Levelt (Ann. Inst. Fourier, Grenoble 23,1 (1973), pp. 157-195) to transform it to a simple pole.

Procedures:

`detadj(U)`
: determinant and adjoint matrix of square matrix U

`invunit(u,n)`
: series inverse of polynomial u up to order n

`jacoblift(f)`
: lifts f^kappa in jacob(f) with minimal kappa

`monodromyB(f[,opt])`
: monodromy of isolated hypersurface singularity f

`H2basis(f)`
: basis of Brieskorn lattice H"

See also: Section D.6.10 [gmspoly_lib], page 779; Section D.6.9 [gmssing_lib], page 778.

D.6.14 qhmoduli_lib

Library: qhmoduli.lib

Purpose: Moduli Spaces of Semi-Quasihomogeneous Singularities

Author: Thomas Bayer, email: bayert@in.tum.de

Procedures:

`ArnoldAction(f, [G, w])`
: Induced action of G_f on T_.

`ModEqn(f)`
: Equations of the moduli space for principal part f

`QuotientEquations(G,A,I)`
: Equations of Variety(I)/G w.r.t. action 'A'

`StabEqn(f)`
: Equations of the stabilizer of f.

`StabEqnId(I, w)`
: Equations of the stabilizer of the qhom. ideal I.

`StabOrder(f)`
: Order of the stabilizer of f.

`UpperMonomials(f, [w])`
: Upper basis of the Milnor algebra of f.

`Max(data)`
: maximal integer contained in 'data'

`Min(data)`
: minimal integer contained in 'data'

D.6.15 sing_lib

Library: sing.lib

Purpose: Invariants of Singularities

Authors: Gert-Martin Greuel, email: greuel@mathematik.uni-kl.de
Bernd Martin, email: martin@math.tu-cottbus.de

Procedures:

`codim(id1, id2)`
: vector space dimension of id2/id1 if finite

`deform(i)`
: infinitesimal deformations of ideal i

`dim_slocus(i)`
: dimension of singular locus of ideal i

`is_active(f,id)`
: is polynomial f an active element mod id? (id ideal/module)

`is_ci(i)` is ideal i a complete intersection?

`is_is(i)` is ideal i an isolated singularity?

`is_reg(f,id)`
: is polynomial f a regular element mod id? (id ideal/module)

`is_regs(i[,id])`
: are gen's of ideal i regular sequence modulo id?

`locstd(i)`
: SB for local degree ordering without cancelling units

`milnor(i)`
: milnor number of ideal i; (assume i is ICIS in nf)

`nf_icis(i)`
: generic combinations of generators; get ICIS in nf

`slocus(i)`
: ideal of singular locus of ideal i

`qhspectrum(f,w)`
: spectrum numbers of w-homogeneous polynomial f

`Tjurina(i)`
: SB of Tjurina module of ideal i (assume i is ICIS)

`tjurina(i)`
: Tjurina number of ideal i (assume i is ICIS)

`T_1(i)` T^1-module of ideal i

`T_2((i)` T^2-module of ideal i

`T_12(i)` T^1- and T^2-module of ideal i

`tangentcone(id)`
compute tangent cone of id

D.6.16 spcurve_lib

Library: spcurve.lib

Purpose: Deformations and Invariants of CM-codim 2 Singularities

Author: Anne Fruehbis-Krueger, anne@mathematik.uni-kl.de

Procedures:

`isCMcod2(i)`
presentation matrix of the ideal i, if i is CM

`CMtype(i)`
Cohen-Macaulay type of the ideal i

`matrixT1(M,n)`
1st order deformation T1 in matrix description

`semiCMcod2(M,T1)`
semiuniversal deformation of maximal minors of M

`discr(sem,n)`
discriminant of semiuniversal deformation

`qhmatrix(M)`
weights if M is quasihomogeneous

`relweight(N,W,a)`
relative matrix weight of N w.r.t. weights (W,a)

`posweight(M,T1,i)`
deformation of coker(M) of non-negative weight

`KSpencerKernel(M)`
kernel of the Kodaira-Spencer map

D.6.17 spectrum_lib

Library: spectrum.lib

Purpose: Singularity Spectrum for Nondegenerate Singularities

Author: S. Endrass

Procedures:

`spectrumnd(poly f[,1])`
spectrum of nondegenerate isolated singularity f

D.6.18 surfacesignature_lib

Library: surfacesignature.lib

Purpose: signature of surface singularity

Authors: Gerhard Pfister pfister@mathematik.uni-kl.de
Muhammad Ahsan Banyamin ahsanbanyamin@gmail.com
Stefan Steidel steidel@mathematik.uni-kl.de

Overview: A library for computing the signature of irreducible surface singularity. The signature of a surface singularity is defined in [3]. The algorithm we use has been proposed in [9].
Let g in C[x,y] define an isolated curve singularity at 0 in C^2 and f:=z^N+g(x,y). The zero-set V:=V(f) in C^3 of f has an isolated singularity at 0. For a small e>0 let V_e:=V(f-e) in C^3 be the Milnor fibre of (V,0) and s: H_2(V_e,R) x H_2(V_e,R) —> R be the intersection form (cf. [1],[7]). H_2(V_e,R) is an m-dimensional R-vector space, m the Milnor number of (V,0) (cf. [1],[4],[5],[6]), and s is a symmetric bilinear form. Let sigma(f) be the signature of s, called the signature of the surface singularity (V,0). Formulaes to compute the signature are given by Nemethi (cf. [8],[9]) and van Doorn, Steenbrink (cf. [2]).
We have implemented three approaches using Puiseux expansions, the resolution of singularities resp. the spectral pairs of the singularity.

References:

[1] Arnold, V.I.; Gusein-Zade, S.M.; Varchenko, A.N.: Singularities of Differentiable Mappings. Vol. 1,2, Birkh"auser (1988). [2] van Doorn, M.G.M.; Steenbrink, J.H.M.: A supplement to the monodromy theorem. Abh. Math. Sem. Univ. Hamburg 59, 225-233 (1989). [3] Durfee, A.H.: The Signature of Smoothings of Complex Surface Singularities. Mathematische Annalen 232, 85-98 (1978). [4] de Jong, T.; Pfister, G.: Local Analytic Geometry. Vieweg (2000). [5] Kerner, D.; Nemethi, A.: The Milnor fibre signature is not semi-continous. arXiv:0907.5252 (2009).
[6] Kulikov, V.S.: Mixed Hodge Structures and Singularities. Cambridge Tracts in Mathematics 132, Cambridge University Press (1998). [7] Nemethi, A.: The real Seifert form and the spectral pairs of isolated hypersurface singularities. Compositio Mathematica 98, 23-41 (1995). [8] Nemethi, A.: Dedekind sums and the signature of f(x,y)+z^N. Selecta Mathematica, New series, Vol. 4, 361-376 (1998).
[9] Nemethi, A.: The Signature of f(x,y)+z^$. Proceedings of Real and Complex Singularities (C.T.C. Wall's 60th birthday meeting, Liverpool (England), August 1996), London Math. Soc. Lecture Notes Series 263, 131–149 (1999).

Procedures:

`signatureBrieskorn(a1,a2,a3)`
signature of singularity x^a1+y^a2+z^a3

`signaturePuiseux(N,f)`
signature of singularity z^N+f(x,y)=0, f irred.

`signatureNemethi(N,f)`
signature of singularity z^N+f(x,y)=0

D.7 Invariant theory

D.7.1 finvar_lib

Library: finvar.lib

Purpose: Invariant Rings of Finite Groups

Author: Agnes E. Heydtmann, contact via Wolfram Decker: decker@mathematik.uni-kl.de Simon A. King, email: simon.king@nuigalway.ie

Overview: A library for computing polynomial invariants of finite matrix groups and generators of related varieties. The algorithms are based on B. Sturmfels, G. Kemper, S. King and W. Decker et al..

Procedures:

`invariant_ring()`
: generators of the invariant ring (i.r.)

`invariant_ring_random()`
: generators of the i.r., randomized alg.

`primary_invariants()`
: primary invariants (p.i.)

`primary_invariants_random()`
: primary invariants, randomized alg.

`invariant_algebra_reynolds()`
: minimal generating set for the invariant ring of a finite matrix group, in the non-modular case

`invariant_algebra_perm()`
: minimal generating set for the invariant ring or a permutation group, in the non-modular case

`cyclotomic()`
: cyclotomic polynomial

`group_reynolds()`
: finite group and Reynolds operator (R.o.)

`molien()` Molien series (M.s.)

`reynolds_molien()`
: Reynolds operator and Molien series

`partial_molien()`
: partial expansion of Molien series

`evaluate_reynolds()`
: image under the Reynolds operator

`invariant_basis()`
: basis of homogeneous invariants of a degree

`invariant_basis_reynolds()`
: as invariant_basis(), with R.o.

`primary_char0()`
: primary invariants (p.i.) in char 0

`primary_charp()`
primary invariants in char p

`primary_char0_no_molien()`
p.i., char 0, without Molien series

`primary_charp_no_molien()`
p.i., char p, without Molien series

`primary_charp_without()`
p.i., char p, without R.o. or Molien series

`primary_char0_random()`
primary invariants in char 0, randomized

`primary_charp_random()`
primary invariants in char p, randomized

`primary_char0_no_molien_random()`
p.i., char 0, without M.s., randomized

`primary_charp_no_molien_random()`
p.i., char p, without M.s., randomized

`primary_charp_without_random()`
p.i., char p, without R.o. or M.s., random.

`power_products()`
exponents for power products

`secondary_char0()`
secondary invariants (s.i.) in char 0

`irred_secondary_char0()`
irreducible s.i. in char 0

`secondary_charp()`
s.i. in char p, with Molien series and Reynolds operator

`secondary_no_molien()`
s.i., without Molien series but with Reynolds operator

`irred_secondary_no_molien()`
irreducible s.i., without Molien series but with Reynolds operator

`secondary_and_irreducibles_no_molien()`
s.i. & irreducible s.i., without M.s.

`secondary_not_cohen_macaulay()`
s.i. when the invariant ring is not Cohen-Macaulay

`orbit_variety()`
ideal of the orbit variety

`rel_orbit_variety()`
ideal of a relative orbit variety (new version)

`relative_orbit_variety()`
ideal of a relative orbit variety (old version)

`image_of_variety()`
ideal of the image of a variety orbit_sums orbit sums of a set of monomials under the action of a permutation group

D.7.2 ainvar_lib

Library: ainvar.lib

Purpose: Invariant Rings of the Additive Group

Authors: Gerhard Pfister (email: pfister@mathematik.uni-kl.de), Gert-Martin Greuel (email: greuel@mathematik.uni-kl.de)

Procedures:

`invariantRing(m..)`
compute ring of invariants of (K,+)-action given by m

`derivate(m,f)`
derivation of f with respect to the vector field m

`actionIsProper(m)`
tests whether action defined by m is proper

`reduction(p,I)`
SAGBI reduction of p in the subring generated by I

`completeReduction()`
complete SAGBI reduction

`localInvar(m,p..)`
invariant polynomial under m computed from p,...

`furtherInvar(m..)`
compute further invariants of m from the given ones

`sortier(id)`
sorts generators of id by increasing leading terms

D.7.3 rinvar_lib

Library: rinvar.lib

Purpose: Invariant Rings of Reductive Groups

Author: Thomas Bayer, tbayer@in.tum.de http://wwwmayr.informatik.tu-muenchen.de/personen/bayert/ Current Address: Institut fuer Informatik, TU Muenchen

Overview: Implementation based on Derksen's algorithm. Written in the scope of the diploma thesis (advisor: Prof. Gert-Martin Greuel) 'Computations of moduli spaces of semiquasihomogenous singularities and an implementation in Singular'

Procedures:

`HilbertSeries(I, w)`
Hilbert series of the ideal I w.r.t. weight w

`HilbertWeights(I, w)`
weighted degrees of the generators of I

`ImageVariety(I, F)`
ideal of the image variety F(variety(I))

`ImageGroup(G, F)`
ideal of G w.r.t. the induced representation

`InvariantRing(G, Gaction)`
: generators of the invariant ring of G

`InvariantQ(f, G, Gaction)`
: decide if f is invariant w.r.t. G

`LinearizeAction(G, Gaction)`
: linearization of the action 'Gaction' of G

`LinearActionQ(action,s,t)`
: decide if action is linear in var(s..nvars)

`LinearCombinationQ(base, f)`
: decide if f is in the linear hull of 'base'

`MinimalDecomposition(f,s,t)`
: minimal decomposition of f (like coef)

`NullCone(G, act)`
: ideal of the nullcone of the action 'act' of G

`ReynoldsImage(RO,f)`
: image of f under the Reynolds operator 'RO'

`ReynoldsOperator(G, Gaction)`
: Reynolds operator of the group G

`SimplifyIdeal(I[,m,s])`
: simplify the ideal I (try to reduce variables)

See also: Section D.6.14 [qhmoduli_lib], page 782; Section D.8.6 [zeroset_lib], page 792.

D.7.4 stratify_lib

Library: stratify.lib

Purpose: Algorithmic Stratification for Unipotent Group-Actions

Author: Anne Fruehbis-Krueger, anne@mathematik.uni-kl.de

Procedures:

`prepMat(M,wr,ws,step)`
: list of submatrices corresp. to given filtration

`stratify(M,wr,ws,step)`
: algorithmic stratifcation (main procedure)

D.8 Symbolic-numerical solving

D.8.1 ffsolve_lib

Library: ffsolve.lib

Purpose: multivariate equation solving over finite fields

Author: Gergo Gyula Borus, borisz@borisz.net

Procedures:

`ffsolve()`
  finite field solving using heuristically chosen method

`PEsolve()`
  solve system of multivariate equations over finite field

`simplesolver()`
  solver using modified exhausting search

`GBsolve()`
  multivariate solver using Groebner-basis

`XLsolve()`
  multivariate polynomial solver using linearization

`ZZsolve()`
  solve system of multivariate equations over finite field

D.8.2 presolve_lib

Library: presolve.lib

Purpose: Pre-Solving of Polynomial Equations

Author: Gert-Martin Greuel, email: greuel@mathematik.uni-kl.de,

Procedures:

`degreepart(id,d1,d2)`
  elements of id of total degree >= d1 and <= d2, and rest

`elimlinearpart(id)`
  linear part eliminated from id

`elimpart(id[,n])`
  partial elimination of vars [among first n vars]

`elimpartanyr(i,p)`
  factors of p partially eliminated from i in any ring

`fastelim(i,p[..])`
  fast elimination of factors of p from i [options]

`findvars(id)`
  variables occuring/not occurring in id

`hilbvec(id[,c,o])`
  intvec of Hilberseries of id [in char c and ord o]

`linearpart(id)`
  elements of id of total degree <=1

`tolessvars(id[,])`
  maps id to new basering having only vars occuring in id

`solvelinearpart(id)`
  reduced std-basis of linear part of id

`sortandmap(id[..])`
  map to new basering with vars sorted w.r.t. complexity

`sortvars(id[n1,p1..])`
sort vars w.r.t. complexity in id [different blocks]

`valvars(id[..])`
valuation of vars w.r.t. to their complexity in id

`idealSplit(id,tF,fS)`
a list of ideals such that their intersection has the same radical as id

D.8.3 solve_lib

Library: solve.lib

Purpose: Complex Solving of Polynomial Systems

Author: Moritz Wenk, email: wenk@mathematik.uni-kl.de
Wilfred Pohl, email: pohl@mathematik.uni-kl.de

Procedures:

`laguerre_solve(p,[..])`
find all roots of univariate polynomial p

`solve(i,[..])`
all roots of 0-dim. ideal i using triangular sets

`ures_solve(i,[..])`
find all roots of 0-dimensional ideal i with resultants

`mp_res_mat(i,[..])`
multipolynomial resultant matrix of ideal i

`interpolate(p,v,d)`
interpolate polynomial from evaluation points i and results j

`fglm_solve(i,[..])`
find roots of 0-dim. ideal using FGLM and lex_solve

`lex_solve(i,p,[..])`
find roots of reduced lexicographic standard basis

`simplexOut(l)`
prints solution of simplex in nice format

`triangLf_solve(l,[..])`
find roots using triangular sys. (factorizing Lazard)

`triangM_solve(l,[..])`
find roots of given triangular system (Moeller)

`triangL_solve(l,[..])`
find roots using triangular system (Lazard)

`triang_solve(l,p,[..])`
find roots of given triangular system

D.8.4 triang_lib

Library: triang.lib

Purpose: Decompose Zero-dimensional Ideals into Triangular Sets

Author: D. Hillebrand

Procedures:

`triangL(G)`
: Decomposition of (G) into triangular systems (Lazard).

`triangLfak(G)`
: Decomp. of (G) into tri. systems plus factorization.

`triangM(G[,.])`
: Decomposition of (G) into triangular systems (Moeller).

`triangMH(G[,.])`
: Decomp. of (G) into tri. syst. with disjoint varieties.

D.8.5 ntsolve_lib

Library: ntsolve.lib

Purpose: Real Newton Solving of Polynomial Systems

Authors: Wilfred Pohl, email: pohl@mathematik.uni-kl.de
Dietmar Hillebrand

Procedures:

`nt_solve(G,ini,[..])`
: find one real root of 0-dimensional ideal G

`triMNewton(G,a,[..])`
: find one real root for 0-dim triangular system G

D.8.6 zeroset_lib

Library: zeroset.lib

Purpose: Procedures for roots and factorization

Author: Thomas Bayer, email: tbayer@mathematik.uni-kl.de,
http://wwwmayr.informatik.tu-muenchen.de/personen/bayert/
Current address: Hochschule Ravensburg-Weingarten

Overview: Algorithms for finding the zero-set of a zero-dim. ideal in Q(a)[x_1,..,x_n], roots and factorization of univariate polynomials over Q(a)[t] where a is an algebraic number. Written in the scope of the diploma thesis (advisor: Prof. Gert-Martin Greuel) 'Computations of moduli spaces of semiquasihomogeneous singularities and an implementation in Singular'. This library is meant as a preliminary extension of the functionality of Singular for univariate factorization of polynomials over simple algebraic extensions in characteristic 0.

Note: Subprocedures with postfix 'Main' require that the ring contains a variable 'a' and no parameters, and the ideal 'mpoly', where 'minpoly' from the basering is stored.

Procedures:

`Quotient(f, g)`
: quotient q of f w.r.t. g (in f = q*g + remainder)

`remainder(f,g)`
: remainder of the division of f by g

`roots(f)`
: computes all roots of f in an extension field of Q

`sqfrNorm(f)`
: norm of f (f must be squarefree)

`zeroSet(I)`
: zero-set of the 0-dim. ideal I

`egcdMain(f, g)`
: gcd over an algebraic extension field of Q

`factorMain(f)`
: factorization of f over an algebraic extension field

`invertNumberMain(c)`
: inverts an element of an algebraic extension field

`quotientMain(f, g)`
: quotient of f w.r.t. g

`remainderMain(f,g)`
: remainder of the division of f by g

`rootsMain(f)`
: computes all roots of f, might extend the ground field

`sqfrNormMain(f)`
: norm of f (f must be squarefree)

`containedQ(data, f)`
: f in data ?

`sameQ(a, b)`
: a == b (list a,b)

D.8.7 signcond_lib

Library: signcond.lib

Purpose: Routines for computing realizable sign conditions

Author: Enrique A. Tobis, etobis@dc.uba.ar

Overview: Routines to determine the number of solutions of a multivariate polynomial system which satisfy a given sign configuration.

References:
Basu, Pollack, Roy, "Algorithms in Real Algebraic Geometry", Springer, 2003.

Procedures:

`signcnd(P,I)`
: The sign conditions realized by polynomials of P on a V(I)

`psigncnd(P,l)`
: Pretty prints the output of signcnd (l)

`firstoct(I)`
: The number of elements of V(I) with every coordinate > 0

D.9 Visualization

D.9.1 graphics_lib

Library: graphics.lib

Purpose: Procedures to use Graphics with Mathematica

Author: Christian Gorzel, gorzelc@math.uni-muenster.de

Procedures:

`staircase(fname,I)`
: Mathematica text for displaying staircase of I

`mathinit()`
: string for loading Mathematica's ImplicitPlot

`mplot(fname,I[# s])`
: Mathematica text for various plots

D.9.2 latex_lib

Library: latex.lib

Purpose: Typesetting of Singular-Objects in LaTeX2e

Author: Christian Gorzel, gorzelc@math.uni-muenster.de

Global variables:

TeXwidth, TeXnofrac, TeXbrack, TeXproj, TeXaligned, TeXreplace, NoDollars are used to control the typesetting. Call `texdemo();` to obtain a LaTeX2e file `texlibdemo.tex` explaining the features of `latex.lib` and its global variables.

`TeXwidth` (int) -1, 0, 1..9, >9: controls breaking of long polynomials
`TeXnofrac` (int) flag: write 1/2 instead of \frac{1}{2}
`TeXbrack` (string) "{", "(", "<", "|", empty string:
controls brackets around ideals and matrices
`TeXproj` (int) flag: write ":" instead of "," in vectors
`TeXaligned` (int) flag: write maps (and ideals) aligned
`TeXreplace` (list) list entries = 2 strings: replacing symbols
`NoDollars` (int) flag: suppresses surrounding $ signs

Procedures:

`closetex(fnm)`
: writes closing line for LaTeX-document

`opentex(fnm)`
: writes header for LaTeX-file fnm

`tex(fnm)` calls LaTeX2e for LaTeX-file fnm

`texdemo([n])`
: produces a file explaining the features of this lib

`texfactorize(fnm,f)`
: creates string in LaTeX-format for factors of polynomial f

`texmap(fnm,m,r1,r2)`
: creates string in LaTeX-format for map m:r1->r2

`texname(fnm,s)`
creates string in LaTeX-format for identifier

`texobj(l)`
creates string in LaTeX-format for any (basic) type

`texpoly(f,n[,l])`
creates string in LaTeX-format for poly

`texproc(fnm,p)`
creates string in LaTeX-format of text from proc p

`texring(fnm,r[,l])`
creates string in LaTeX-format for ring/qring

`rmx(s)` removes .aux and .log files of LaTeX-files

`xdvi(s)` calls xdvi for dvi-files (parameters in square brackets [] are optional) (Procedures with file output assume sufficient write permissions when trying to append existing or create new files.)

D.9.3 surf_lib

Library: surf.lib

Purpose: Procedures for Graphics with Surf

Author: Hans Schoenemann, Frank Seelisch

Note: Using this library requires the program `surf` to be installed. You can download `surf` either from `http://sourceforge.net/projects/surf` or from `ftp://www.mathematik.uni-kl.de/pub/Math/Singular/utils/`. The procedure surfer requires the program `surfer` (version 1.4.1 or newwer) to be installed. You can download `surfer` from `http://imaginary.org/program/surfer`

Procedures:

`plot(I)` plots plane curves and surfaces

`surfer(I)`
plots surfaces interactivly

See also: Section D.9.4 [surfex_lib], page 795.

D.9.4 surfex_lib

Library: surfex.lib

Purpose: Procedures for visualizing and rotating surfaces.

Author: Oliver Labs
This library uses the program surf
(written by Stefan Endrass and others)
and surfex (written by Oliver Labs and others, mainly Stephan Holzer).

Note: It is still an alpha version (see `http://www.AlgebraicSurface.net`)
This library requires the program surfex, surf and java to be installed. The software is used for producing raytraced images of surfaces. You can download `surfex` from `http://www.surfex.AlgebraicSurface.net`

surfex is a front-end for surf which aims to be easier to use than the original tool.

Procedures:

`plotRotated(poly,coord)`
: Plot the surface given by the polynomial p with the coordinates coords(list)

`plotRot(poly)`
: Similar to plotRotated, but guesses automatically which coordinates should be used

`plotRotatedList(varieties, coords)`
: Plot the varieties given by the list varieties with the coordinates coords

`plotRotatedDirect(varieties)`
: Plot the varieties given by the list varietiesList

`plotRotatedListFromSpecifyList(varietiesList)`
: Plot the varieties given by the list varietiesList

See also: Section D.9.3 [surf_lib], page 795.

D.10 Coding theory

D.10.1 brnoeth_lib

Library: brnoeth.lib

Purpose: Brill-Noether Algorithm, Weierstrass-SG and AG-codes

Authors: Jose Ignacio Farran Martin, ignfar@eis.uva.es
Christoph Lossen, lossen@mathematik.uni-kl.de

Overview: Implementation of the Brill-Noether algorithm for solving the Riemann-Roch problem and applications to Algebraic Geometry codes. The computation of Weierstrass semigroups is also implemented.
The procedures are intended only for plane (singular) curves defined over a prime field of positive characteristic.
For more information about the library see the end of the file brnoeth.lib.

Procedures:

`Adj_div(f)`
: computes the conductor of a curve

`NSplaces(h,A)`
: computes non-singular places with given degrees

`BrillNoether(D,C)`
: computes a vector space basis of the linear system L(D)

`Weierstrass(P,m,C)`
: computes the Weierstrass semigroup of C at P up to m

`extcurve(d,C)`
: extends the curve C to an extension of degree d

`AGcode_L(G,D,E)`
: computes the evaluation AG code with divisors G and D

`AGcode_Omega(G,D,E)`
: computes the residual AG code with divisors G and D

`prepSV(G,D,F,E)`
preprocessing for the basic decoding algorithm

`decodeSV(y,K)`
decoding of a word with the basic decoding algorithm

`closed_points(I)`
computes the zero-set of a zero-dim. ideal in 2 vars

`dual_code(C)`
computes the dual code

`sys_code(C)`
computes an equivalent systematic code

`permute_L(L,P)`
applies a permutation to a list

D.10.2 decodegb_lib

Library: decodegb.lib

Purpose: Decoding and min distance of linear codes with GB

Author: Stanislav Bulygin, bulygin@mathematik.uni-kl.de

Overview: In this library we generate several systems used for decoding cyclic codes and finding their minimum distance. Namely, we work with the Cooper's philosophy and generalized Newton identities. The origindeal method of quadratic equations is worked out here as well. We also (for comparison) enable to work with the system of Fitzgerald-Lax. We provide some auxiliary functions for further manipulations and decoding. For an overview of the methods mentioned above Section C.8 [Decoding codes with Groebner bases], page 712. For the vanishing ideal computation the algorithm of Farr and Gao is implemented.

Procedures:

`sysCRHT(..)`
generates the CRHT-ideal as in Cooper's philosophy

`sysCRHTMindist(..)`
CRHT-ideal to find the minimum distance in the binary case

`sysNewton(..)`
generates the ideal with the generalized Newton identities

`sysBin(..)`
generates Bin system using Waring function

`encode(x,g)`
encodes given message x with the given generator matrix g

`syndrome(h,c)`
computes a syndrome w.r.t. the given check matrix

`sysQE(..)`
generates the system of quadratic equations for decoding

`errorInsert(..)`
inserts errors in a word

`errorRand(y,num,e)`
inserts random errors in a word

`randomCheck(m,n,e)`
generates a random check matrix

`genMDSMat(n,a)`
generates an MDS (actually an RS) matrix

`mindist(check)`
computes the minimum distance of a code

`decode(rec)`
decoding of a word using the system of quadratic equations

`decodeRandom(..)`
a procedure for manipulation with random codes

`decodeCode(..)`
a procedure for manipulation with the given code

`vanishId(points)`
computes the vanishing ideal for the given set of points

`sysFL(..)`
generates the Fitzgerald-Lax system

`decodeRandomFL(..)`
manipulation with random codes via Fitzgerald-Lax

D.11 System and Control theory

D.11.1 Control theory background

Control systems are usually described by differential (or difference) equations, but their properties of interest are most naturally expressed in terms of the system trajectories (the set of all solutions to the equations). This is formalized by the notion of the system *behavior*. On the other hand, the manipulation of linear system equations can be formalized using algebra, more precisely module theory. The relationship between modules and behaviors is very rich and leads to deep results on system structure.

The key to the module-behavior correspondence is a property of some signal spaces that are modules over the ring of differential (or difference) operators, namely, *the injective cogenerator property*. This property makes it possible to translate any statement on the solution spaces that can be expressed in terms of images and kernels, to an equivalent statement on the modules. Thus analytic properties can be identified with algebraic properties, and conversely, the results of manipulating the modules using computer algebra can be re-translated and interpreted using the language of systems theory. This duality (*algebraic analysis*) is widely used in behavioral systems and control theory today.

For instance, a system is **controllable** (a fundamental property for any control system) if and only if the associated module is torsion-free. This concept can be refined by the so-called controllability degrees. The strongest form of controllability (*flatness*) corresponds to a projective (or even free) module.

Controllability means that one can switch from one system trajectory to another without violating the system law (concatenation of trajectories). For one-dimensional systems (ODE) that evolve in time, this is usually interpreted as switching from a given past trajectory to a desired future trajectory. Thus the system can be forced to behave in an arbitrarily prescribed way.

The extreme case opposed to controllability is **autonomy**: autonomous systems evolve independently according to their law, without being influenceable from the outside. Again, the property can be refined in terms of autonomy degrees.

D.11.2 control_lib

Library: control.lib

Purpose: Algebraic analysis tools for System and Control Theory

Authors: Oleksandr Iena, yena@mathematik.uni-kl.de
Markus Becker, mbecker@mathematik.uni-kl.de
Viktor Levandovskyy, levandov@mathematik.uni-kl.de

Support: Forschungsschwerpunkt 'Mathematik und Praxis' (Project of Dr. E. Zerz and V. Levandovskyy), University of Kaiserslautern

Procedures:

`control(R)`
analysis of controllability-related properties of R (using Ext modules)

`controlDim(R)`
analysis of controllability-related properties of R (using dimension)

`autonom(R)`
analysis of autonomy-related properties of R (using Ext modules)

`autonomDim(R)`
analysis of autonomy-related properties of R (using dimension)

`leftKernel(R)`
a left kernel of R

`rightKernel(R)`
a right kernel of R

`leftInverse(R)`
a left inverse of R

`rightInverse(R)`
a right inverse of R

`colrank(M)`
a column rank of M as of matrix

`genericity(M)`
analysis of the genericity of parameters

`canonize(L)`
Groebnerification for modules in the output of control or autonomy procs

`iostruct(R)`
computes an IO-structure of behavior given by a module R

`findTorsion(R, I)`
generators of the submodule of a module R, annihilated by the ideal I

`controlExample(s)`
set up an example from the mini database inside of the library

`view()` well-formatted output of lists, modules and matrices

D.11.3 jacobson_lib

Library: jacobson.lib

Purpose: Algorithms for Smith and Jacobson Normal Form

Author: Kristina Schindelar, Kristina.Schindelar@math.rwth-aachen.de,
Viktor Levandovskyy, levandov@math.rwth-aachen.de

Overview: We work over a ring R, that is an Euclidean principal ideal domain. If R is commutative, we suppose R to be a polynomial ring in one variable. If R is non-commutative, we suppose R to have two variables, say x and d. We treat then the basering as the Ore localization of R with respect to the mult. closed set S = K[x] without 0. Thus, we treat basering as principal ideal ring with d a polynomial variable and x an invertible one.
Note, that in computations no division by x will actually happen.

Given a rectangular matrix M over R, one can compute unimodular (that is invertible) square matrices U and V, such that U*M*V=D is a diagonal matrix. Depending on the ring, the diagonal entries of D have certain properties.

We call a square matrix D as before 'a weak Jacobson normal form of M'. It is known, that over the first rational Weyl algebra K(x)<d>, D can be further transformed into a diagonal matrix (1,1,...,1,f,0,..,0), where f is in K(x)<d>. We call such a form of D the strong Jacobson normal form. The existence of strong form in not guaranteed if one works with algebra, which is not rational Weyl algebra.

References:

[1] N. Jacobson, 'The theory of rings', AMS, 1943.
[2] Manuel Avelino Insua Hermo, 'Varias perspectives sobre las bases de Groebner : Forma normal de Smith, Algorithme de Berlekamp y algebras de Leibniz'.
PhD thesis, Universidad de Santiago de Compostela, 2005.
[3] V. Levandovskyy, K. Schindelar 'Computing Jacobson normal form using Groebner bases',
to appear in Journal of Symbolic Computation, 2010.

Procedures:

`smith(M[,eng1,eng2])`
: compute the Smith Normal Form of M over commutative ring

`jacobson(M[,eng])`
: compute a weak Jacobson Normal Form of M over non-commutative ring

`divideUnits(L)`
: create ones out of units in the output of smith or jacobson

See also: Section D.11.2 [control_lib], page 799.

D.11.4 findifs_lib

Library: findifs.lib

Purpose: Tools for the finite difference schemes

Authors: Viktor Levandovskyy, levandov@math.rwth-aachen.de

Overview: We provide the presentation of difference operators in a polynomial, semi-factorized and a nodal form. Running `findifs_example();` will demonstrate, how we generate finite difference schemes of linear PDEs from given approximations.

Theory: The method we use have been developed by V. Levandovskyy and Bernd Martin. The computation of a finite difference scheme of a given single linear partial differential equation with constant coefficients with a given approximation rules boils down to the computation of a Groebner basis of a submodule of a free module with respect to the ordering, eliminating module components.

Support: SpezialForschungsBereich F1301 of the Austrian FWF

Procedures:

`findifs_example()`
: containes a guided explanation of our approach

`decoef(P,n)`
: decompose polynomial P into summands with respect to the number n

`difpoly2tex(S,P[,Q])`
: present the difference scheme in the nodal form

`exp2pt(P[,L])`
: convert a polynomial M into the TeX format, in nodal form

`texcoef(n)`
: converts the number n into TeX

`npar(n)`
: search for 'n' among the parameters and returns its number

`magnitude(P)`
: compute the square of the magnitude of a complex expression

`replace(s,what,with)`
: replace in s all the substrings with a given string

`xchange(w,a,b)`
: exchange two substrings in a given string

See also: Section D.15.13 [finitediff_lib], page 838; Section D.9.2 [latex_lib], page 794.

D.12 Teaching

The libraries in this section are intended to be used for teaching purposes but not for serious computations.

D.12.1 aksaka_lib

Library: aksaka.lib

Purpose: Procedures for primality testing after Agrawal, Saxena, Kayal

Authors: Christoph Mang

Overview: Algorithms for primality testing in polynomial time based on the ideas of Agrawal, Saxena and Kayal.

Procedures:

`fastExpt(a,m,n)`
: `a^m` for numbers a,m; if `a^k>n n+1` is returned

`log2(n)` logarithm to basis 2 of n

`PerfectPowerTest(n)`
checks if there are a,b>1, so that a^b=n

`wurzel(r)`
square root of number r

`euler(r)` phi-function of Euler

`coeffmod(f,n)`
polynomial f modulo number n (coefficients mod n)

`powerpolyX(q,n,a,r)`
(polynomial a)^q modulo (poly r,number n)

`ask(n)` ASK-Algorithm; deterministic Primality test

D.12.2 atkins_lib

Library: atkins.lib

Purpose: Procedures for teaching cryptography

Author: Stefan Steidel, steidel@mathematik.uni-kl.de

Note: The library contains auxiliary procedures to compute the elliptic curve primality test of Atkin and the Atkin's Test itself. The library is intended to be used for teaching purposes but not for serious computations. Sufficiently high printlevel allows to control each step, thus illustrating the algorithms at work.

Procedures:

`newTest(L,D)`
checks if number D already exists in list L

`bubblesort(L)`
sorts elements of the list L

`disc(N,k)`
generates a list of negative discriminants

`Cornacchia(d,p)`
computes solution (x,y) for x^2+d*y^2=p

`CornacchiaModified(D,p)`
computes solution (x,y) for x^2+|D|*y^2=4p

`maximum(L)`
computes the maximal number contained in L

`sqr(w,k)` computes the square root of w w.r.t. accuracy k

`expo(z,k)`
computes exp(z)

`jOft(t,k)`
computes the j-invariant of t

`round(r)` rounds r to the nearest number out of Z

`HilbertClassPoly(D,k)`
computes the Hilbert Class Polynomial

`rootsModp(p,P)`
computes roots of the polynomial P modulo p

`wUnit(D)` computes the number of units in Q(sqr(D))

`Atkin(N,K,B)`
tries to prove that N is prime

D.12.3 crypto_lib

Library: crypto.lib

Purpose: Procedures for teaching cryptography

Authors: Gerhard Pfister, pfister@mathematik.uni-kl.de
David Brittinger, dativ@gmx.net

Overview: The library contains procedures to compute the discrete logarithm, primality-tests, factorization included elliptic curves. The library is intended to be used for teaching purposes but not for serious computations. Sufficiently high printlevel allows to control each step, thus illustrating the algorithms at work.

Procedures:

`decimal(s)`
number corresponding to the hexadecimal number s

`eexgcdN(L)`
T with sum_i L[i]*T[i]=T[n+1]=gcd(L[1],...,L[n])

`lcmN(a,b)`
compute lcm(a,b)

`powerN(m,d,n)`
compute m^d mod n

`chineseRem(T,L)`
compute x such that x = T[i] mod L[i]

`Jacobi(a,n)`
the generalized Legendre symbol of a and n

`primList(n)`
the list of all primes <=n

`primL(q)` first primes p_1,...,p_r such that q<p_1*...*p_r

`intPart(x)`
the integral part of a rational number

`intRoot(m)`
the integral part of the square root of m

`squareRoot(a,p)`
the square root of a in Z/p, p prime

`solutionsMod2(M)`
basis solutions of Mx=0 over Z/2

`powerX(q,i,I)`
q-th power of the i-th variable modulo I

`babyGiant(b,y,p)`
discrete logarithm x: b^x=y mod p

`rho(b,y,p)`
discrete logarithm x: b^x=y mod p

`MillerRabin(n,k)`
probabilistic primaly-test of Miller-Rabin

`SolowayStrassen(n,k)`
probabilistic primaly-test of Soloway-Strassen

`PocklingtonLehmer(N,[])`
primaly-test of Pocklington-Lehmer

`PollardRho(n,k,a,[])`
Pollard's rho factorization

`pFactor(n,B,P)`
Pollard's p-factorization

`quadraticSieve(n,c,B,k)`
quadratic sieve factorization

`isOnCurve(N,a,b,P)`
P is on the curve y^2z=x^3+a*xz^2+b*z^3 over Z/N

`ellipticAdd(N,a,b,P,Q)`
P+Q, addition on elliptic curves

`ellipticMult(N,a,b,P,k)`
k*P on elliptic curves

`ellipticRandomCurve(N)`
generates y^2z=x^3+a*xz^2+b*z^3 over Z/N randomly

`ellipticRandomPoint(N,a,b)`
random point on y^2z=x^3+a*xz^2+b*z^3 over Z/N

`countPoints(N,a,b)`
number of points of y^2=x^3+a*x+b over Z/N

`ellipticAllPoints(N,a,b)`
points of y^2=x^3+a*x+b over Z/N

`ShanksMestre(q,a,b,[])`
number of points of y^2=x^3+a*x+b over Z/N

`Schoof(N,a,b)`
number of points of y^2=x^3+a*x+b over Z/N

`generateG(a,b,m)`
m-th division polynomial of y^2=x^3+a*x+b over Z/N

`factorLenstraECM(N,S,B,[])`
Lenstra's factorization

`ECPP(N)` primaly-test of Goldwasser-Kilian

`calculate_ordering(num1, primitive, mod1)`
Calculates x so that primitive^x == num1 mod mod1

`is_primitive_root(primitive, mod1)`
: Checks if primitive is a primitive root modulo mod1

`find_first_primitive_root(mod1)`
: Returns the first primitive root modulo mod1, starting with 1

`binary_add(binary_list)`
: Adds a 1 to a binary encoded list

`inverse_modulus(num,mod1)`
: Finds a t so that t*num = 1 mod mod1

`is_prime(n)`
: Checks if n is prime proc find_biggest_index(a) Returns the index of the biggest element of a

`find_index(a,e)`
: Returns the list index of element e in list a. Returns 0 if e is not in a

`subset_sum01(list knapsack, int solution)`
: solves the subset-sum-knapsack-problem by calculating all subsets and choosing the right solution

`subset_sum02(list knapsack, int sol)`
: solves the subset-sum-knapsack-problem with a naive greedy algorithm

`unbounded_knapsack(list knapsack, list profit, int capacity)`
: solves the unbounded_knapsack-problem, needing a list of knapsack weights, a list of profits and a capacity

`multidimensional_knapsack(matrix m, list capacities, list profits)`
: solves the multidimensional_knapsack-problem by using the PECH algorithm, needing a weight matrix m, a list of capacities and a list of profits

`naccache_stern_generation(int key, int primenum)`
: generates a hard knapsack for the Naccache-Stern Kryptosystem for given key and prime modulus

`naccache_stern_encryption(list knapsack, list message, int primenum)`
: encrypts a message with the Naccache-Stern Kryptosystem, using a hard knapsack, a message encoded as binary list and a prime modulus

`naccache_stern_decryption(list knapsack, int key, int primenum, int message)`
: decrypts a message with the Naccache-Stern Kryptosystem, using the easy knapsack, the key, the prime modulus and the message encoded as integer

`m_merkle_hellman_transformation(list knapsack, int primitive, int mod1)`
: generates a hard knapsack for the multiplicative Merkle-Hellman Kryptosystem for a given easy knapsack and a primitive root for a modulus mod1

`m_merkle_hellman_encryption(list knapsack, list message)`
: encrypts a message with the multiplicative Merkle-Hellman Kryptosystem, using a hard knapsack and a message encoded as binary list

`m_merkle_hellman_decryption(list knapsack, bigint primitive, bigint mod1, int message)` decrypts a message with the multiplicative Merkle-Hellman Kryptosystem, using the easy knapsack, the key given by the primitive root, the modulus mod1 and the message encoded as integer merkle_hellman_transformation(list knapsack, int key, int mod1 generates a hard knapsack for the Merkle-Hellman Kryptosystem for a given easy knapsack , a multiplicator key and a modulus mod1

`merkle_hellman_encryption(list knapsack, list message)`
: encrypts a message with the Merkle-Hellman Kryptosystem, using a hard knapsack and a message encoded as binary list

`merkle_hellman_decryption(list knapsack, int key, int mod1, int message)`
: decrypts a message with the multiplicative Merkle-Hellman Kryptosystem, using the hard knapsack, the key, the modulus mod1 and the message encoded as integer

`super_increasing_knapsack(int ksize)`
: Creates the smallest super-increasing knapsack of given size ksize

`h_increasing_knapsack(int ksize, int h)`
: Creates the smallest h-increasing knapsack of given size ksize and h

`injective_knapsack(int ksize, int kmaxelement)`
: Creates all list of all injective knapsacks of given size ksize and maximal element kmaxelement

`calculate_max_sum(list a)`
: Calculates the maximal sum of a given knapsack a

`set_is_injective(list a)`
: Checks if knapsack a is injective

`is_h_injective(list a, int h)`
: Checks if knapsack a is h-injective

`is_fix_injective(list a)`
: Checks if knapsack a is fix-injective

`three_elements(list out, int iterations)`
: Creates the smallest injective knapsack with a given injective_knapsack by using the three-elements-algorithm with a given number of iterations

D.12.4 hyperel_lib

Library: hyperel.lib

Author: Markus Hochstetter, markushochstetter@gmx.de

Note: The library provides procedures for computing with divisors in the jacobian of hyperelliptic curves. In addition procedures are available for computing the rational representation of divisors and vice versa. The library is intended to be used for teaching and demonstrating purposes but not for efficient computations.

Procedures:

`ishyper(h,f)`
: test, if y^2+h(x)y=f(x) is hyperelliptic

`isoncurve(P,h,f)`
: test, if point P is on C: y^2+h(x)y=f(x)

`chinrestp(b,moduli)`
compute polynom x, s.t. x=b[i] mod moduli[i]

`norm(a,b,h,f)`
norm of a(x)-b(x)y in IF[C]

`multi(a,b,c,d,h,f)`
(a(x)-b(x)y)*(c(x)-d(x)y) in IF[C] ratrep (P,h,f) returns polynomials a,b, s.t. div(a,b)=P

`divisor(a,b,h,f,[])`
computes divisor of a(x)-b(x)y

`gcddivisor(p,q)`
gcd of the divisors p and q

`semidiv(D,h,f)`
semireduced divisor of the pair of polys D[1], D[2]

`cantoradd(D,Q,h,f)`
adding divisors of the hyperell. curve y^2+h(x)y=f(x)

`cantorred(D,h,f)`
returns reduced divisor which is equivalent to D

`double(D,h,f)`
computes 2*D on y^2+h(x)y=f(x)

`cantormult(m,D,h,f)`
computes m*D on y^2+h(x)y=f(x)

D.12.5 teachstd_lib

Library: teachstd.lib

Purpose: Procedures for teaching standard bases

Author: G.-M. Greuel, greuel@mathematik.uni-kl.de

Note: The library is intended to be used for teaching purposes, but not for serious computations. Sufficiently high printlevel allows to control each step, thus illustrating the algorithms at work. The procedures are implemented exactly as described in the book 'A SINGULAR Introduction to Commutative Algebra' by G.-M. Greuel and G. Pfister (Springer 2002).

Procedures:

`ecart(f)` ecart of f

`tail(f)` tail of f

`sameComponent(f,g)`
test for same module component of lead(f) and lead(g)

`leadmonomial(f)`
leading monomial as polynomial (also for vectors)

`monomialLcm(m,n)`
lcm of monomials m and n as polynomial (also for vectors)

`spoly(f[,1])`
s-polynomial of f [symmetric form]

`minEcart(T,h)`
: element g from T of minimal ecart s.t. LM(g)|LM(h)

`NFMora(i)`
: normal form of i w.r.t Mora algorithm

`prodcrit(f,g[,o])`
: test for product criterion

`chaincrit(f,g,h)`
: test for chain criterion

`pairset(G)`
: pairs form G neither satifying prodcrit nor chaincrit

`updatePairs(P,S,h)`
: pairset P enlarded by not useless pairs (h,f), f in S

`standard(id)`
: standard basis of ideal/module

`localstd(id)`
: local standard basis of id using Lazard's method

D.12.6 weierstr_lib

Library: weierstr.lib

Purpose: Procedures for the Weierstrass Theorems

Author: G.-M. Greuel, greuel@mathematik.uni-kl.de

Procedures:

`weierstrDiv(g,f,d)`
: perform Weierstrass division of g by f up to degree d

`weierstrPrep(f,d)`
: perform Weierstrass preparation of f up to degree d

`lastvarGeneral(f)`
: make f general of finite order w.r.t. last variable

`generalOrder(f)`
: compute integer b s.t. f is x_n-general of order b

D.12.7 rootsmr_lib

Library: rootsmr.lib

Purpose: Counting the number of real roots of polynomial systems

Author: Enrique A. Tobis, etobis@dc.uba.ar

Overview: Routines for counting the number of real roots of a multivariate polynomial system. Two methods are implemented: deterministic computation of the number of roots, via the signature of a certain bilinear form (nrRootsDeterm); and a rational univariate projection, using a pseudorandom polynomial (nrRootsProbab). It also includes a command to verify the correctness of the pseudorandom answer.

References:
Basu, Pollack, Roy, "Algorithms in Real Algebraic Geometry", Springer, 2003.

Procedures:

`nrRootsProbab(I)`
Number of real roots of 0-dim ideal (probabilistic)

`nrRootsDeterm(I)`
Number of real roots of 0-dim ideal (deterministic)

`symsignature(m)`
Signature of the symmetric matrix m

`sturmquery(h,B,I)`
Sturm query of h on V(I)

`matbil(h,B,I)`
Matrix of the bilinear form on R/I associated to h

`matmult(f,B,I)`
Matrix of multiplication by f (m_f) on R/I in the basis B

`tracemult(f,B,I)`
Trace of m_f (B is an ordered basis of R/I)

`coords(f,B,I)`
Coordinates of f in the ordered basis B

`randcharpoly(B,I,n)`
Pseudorandom charpoly of univ. projection, n optional

`verify(p,B,i)`
Verifies the result of randcharpoly

`randlinpoly(n)`
Pseudorandom linear polynomial, n optional

`powersums(f,B,I)`
Powersums of the roots of a char polynomial

`symmfunc(S)`
Symmetric functions from the powersums S

`univarpoly(l)`
Polynomial with coefficients from l

`qbase(i)` Like kbase, but the monomials are ordered

D.12.8 rootsur_lib

Library: rootsur.lib

Purpose: Counting number of real roots of univariate polynomial

Author: Enrique A. Tobis, etobis@dc.uba.ar

Overview: Routines for bounding and counting the number of real roots of a univariate polynomial, by means of several different methods, namely Descartes' rule of signs, the Budan-Fourier theorem, Sturm sequences and Sturm-Habicht sequences. The first two give bounds on the number of roots. The other two compute the actual number of roots of the polynomial. There are several wrapper functions, to simplify the application of the aforesaid theorems and some functions to determine whether a given polynomial is univariate.

References:
Basu, Pollack, Roy, "Algorithms in Real Algebraic Geometry", Springer, 2003.

Procedures:

`isuni(p)` Checks whether a polynomial is univariate

`whichvariable(p)`
The only variable of a univariate monomial (or 0)

`varsigns(p)`
Number of sign changes in a list

`boundBuFou(p,a,b)`
Bound for number of real roots of polynomial p in interval (a,b)

`boundposDes(p)`
Bound for the number of positive real roots of polynomial p

`boundDes(p)`
Bound for the number of real roots of polynomial p

`allrealst(p)`
Checks whether all the roots of a polynomial are real (via Sturm)

`maxabs(p)`
A bound for the maximum absolute value of a root of a poly

`allreal(p)`
Checks whether all the roots of a polynomial are real (via St-Ha)

`sturm(p,a,b)`
Number of real roots of a polynomial on an interval (via Sturm)

`sturmseq(p)`
Sturm sequence of a polynomial

`sturmha(p,a,b)`
Number of real roots of a polynomial in (a,b) (via Sturm-Habicht)

`sturmhaseq(p)`
A Sturm-Habicht Sequence of a polynomial

`reverse(l)`
Reverses a list

`nrroots(p)`
The number of real roots of p

`isparam(p)`
Returns 0 if and only if the polynomial has non-parametric coefficients

D.13 Tropical Geometry

D.13.1 gfan_lib

Library: gfan.lib

Purpose: Interface to gfan and gfanlib for computations in convex geometry

Authors: Anders N. Jensen, email: jensen@imf.au.dk
Yue Ren, email: ren@mathematik.uni-kl.de
Frank Seelisch

Procedures:

`fullSpace(n)`
: cone, the ambient space of dimension n

`origin(n)`
: cone, the origin in an ambient space of dimension n

`positiveOrthant(n)`
: cone, the positive orthant of dimension n

`ambientDimension(c)`
: the dimension of the ambient space the input lives in

`canonicalizeCone(c)`
: a unique representation of the cone c

`codimension(c)`
: the codimension of the input

`coneViaPoints()`
: define a cone

`coneViaInequalities()`
: define a cone

`coneLink(c,w)`
: the link of c around w

`containsAsFace(c,d)`
: is d a face of c

`containsInSupport(c,d)`
: is d contained in c

`containsPositiveVector(c)`
: contains a vector with only positive entries?

`containsRelatively(c,p)`
: p in c?

`convexHull(c1,c2)`
: convex hull

`convexIntersection(c1,c2)`
: convex hull

`dimension(c)`
: diemsion

`dualCone(c)`
: the dual of c

`equations(c)`
: defining equations of c

`faceContaining(c,w)`
: the face of c containing w in its relative interior

`facets(c)`
: the facets of c

`generatorsOfLinealitySpace(c)`
: generators of the lineality space of c

`generatorsOfSpan(c)`
: generators of the span of c

`getLinearForms(c)`
: linear forms previously stored in c

`getMultiplicity(c)`
: multiplicity previously stored in c

`inequalities(c)`
: inequalities of c

`isFullSpace(c)`
: is the entire ambient space?

`isOrigin(c)`
: is the origin?

`isSimplicial(c)`
: is simplicial?

`linealityDimension(c)`
: the dimension of the lineality space of c

`linealitySpace(c)`
: the lineality space of c

`negatedCone(c)`
: the negative of c

`polytopeViaInequalities()`

`polytopeViaPoints()`

`quotientLatticeBasis(c)`
: basis of Z^n intersected with the span of c modulo Z^n intersected with the lineality space of c

`randomPoint(c)`
: a random point in the relative interior of c

`rays(c)` generators of the rays of c

`relativeInteriorPoint(c)`
: point in the relative interior of c

`semigroupGenerator(c)`
: generator of Z^n intersected with c modulo Z^n intersected with the lineality space of c

`setLinearForms(c)`
stores linear forms in c

`setMultiplicity(c)`
stores a multiplicity in c

`span(c)` unique irredundant equations of c

`uniquePoint(c)`
a unique point in c stable under reflections at coordinate hyperplanes

`containsInCollection(f,c)`
f contains c?

`emptyFan(n)`
empty fan in ambient dimension n

`fanViaCones(L)`
fan generated by the cones in L

`fullFan(n)`
full fan in ambient dimension n

`fVector(f)`
the f-Vector of f

`getCone(f,d,i[,o,m])`
the i-th cone in dimensiond in f

`insertCone(f,c[,b])`
inserts the cone c into f

`isCompatible(f,c)`
f and c live in the same ambient space

`isPure(f)`
all maximal cones of f are of the same dimension

`nmaxcones(f)`
number of maximal cones in f

`ncones(f)`
number of cones in f

`numberOfConesOfDimension(f,d[,o,m])`
the number of cones in dimension d

`removeCone(f,c[,b])`
removes the cone c

`dualPolytope(p)`
the dual of p

`newtonPolytope(f)`
convex hull of all exponent vectors of f

`vertices(p)`
vertices of p

D.13.2 polymake_lib

Library: polymake.lib

Purpose: Computations with polytopes and fans, interface to polymake and TOPCOM

Author: Thomas Markwig, email: keilen@mathematik.uni-kl.de
Yue Ren, email: ren@mathematik.uni-kl.de

Warning: Most procedures will not work unless polymake or topcom is installed and if so, they will only work with the operating system LINUX! For more detailed information see IMPORTANT NOTE respectively consult the help string of the procedures.

The conventions used in this library for polytopes and fans, e.g. the length and labeling of their vertices resp. rays, differs from the conventions used in polymake and thus from the conventions used in the polymake extension polymake.so of Singular. We recommend to use the newer polymake.so whenever possible.

Important note:

Even though this is a Singular library for computing polytopes and fans such as the Newton polytope or the Groebner fan of a polynomial, most of the hard computations are NOT done by Singular but by the program
- polymake by Ewgenij Gawrilow, TU Berlin and Michael Joswig, TU Darmstadt (see `http://www.polymake.org/`),
respectively (only in the procedure triangulations) by the program
- topcom by Joerg Rambau, Universitaet Bayreuth (see `http://www.rambau.wm.uni-bayreuth.de/TOPCOM/`);
this library should rather be seen as an interface which allows to use a (very limited) number of options which polymake respectively topcom offers to compute with polytopes and fans and to make the results available in Singular for further computations; moreover, the user familiar with Singular does not have to learn the syntax of polymake or topcom, if the options offered here are sufficient for his purposes.
Note, though, that the procedures concerned with planar polygons are independent of both, polymake and topcom.

Procedures using polymake:

`polymakePolytope()`
computes the vertices of a polytope using polymake

`newtonPolytopeP()`
computes the Newton polytope of a polynomial

`newtonPolytopeLP()`
computes the lattice points of the Newton polytope

Procedures using topcom:

`triangulations()`
computes all triangulations of a marked polytope

`secondaryPolytope()`
computes the secondary polytope of a marked polytope

Procedures conerned with planar polygons:

`cycleLength()`
computes the cycleLength of cycle

`splitPolygon()`
splits a marked polygon into vertices, facets, interior points

`eta()` computes the eta-vector of a triangulation

`findOrientedBoundary()`
computes the boundary of a convex hull

`cyclePoints()`
computes lattice points connected to some lattice point

`latticeArea()`
computes the lattice area of a polygon

`picksFormula()`
computes the ingrediants of Pick's formula for a polygon

`ellipticNF()`
computes the normal form of an elliptic polygon

`ellipticNFDB()`
displays the 16 normal forms of elliptic polygons

Procedures using libpolymake:

`boundaryLatticePoints()`

`ehrhartPolynomialCoeff()`

`fVectorP()`

`facetVertexLatticeDistances()`

`facetWidth()`

`facetWidths()`

`gorensteinIndex()`

`gorensteinVector()`

`hStarVector()`

`hVector()`

`hilbertBasis()`

`interiorLatticePoints()`

`isBounded()`

`isCanonical()`

`isCompressed()`

`isGorenstein()`

`isLatticeEmpty()`

`isNormal()`

`isReflexive()`

`isSmooth()`

`isTerminal()`

`isVeryAmple()`

`latticeCodegree()`

`latticeDegree()`

`latticePoints()`

```
latticeVolume()
maximalFace()
maximalValue()
minimalFace()
minimalValue()
minkowskiSum()
nBoundaryLatticePoints()
nHilbertBasis()
nInteriorLatticePoints()
nLatticePoints()
normalFan()
vertexAdjacencyGraph()
vertexEdgeGraph()
visual()
```

D.13.3 realizationMatroids_lib

Library: realizationMatroids.lib

Purpose: Deciding Relative Realizability for Tropical Fan Curves in 2-Dimensional Matroidal Fans

Authors: Anna Lena Winstel, winstel@mathematik.uni-kl.de

Overview: In tropical geometry, one question to ask is the following: given a one-dimensional balanced polyhedral fan C which is set theoretically contained in the tropicalization trop(Y) of an algebraic variety Y, does there exist a curve X in Y such that trop(X) = C? This equality of C and trop(X) denotes an equality of both, the fans trop(X) and C and their weights on the maximal cones. The relative realization space of C with respect to Y is the space of all algebraic curves in Y which tropicalize to C.

This library provides procedures deciding relative realizability for tropical fan curves, i.e. one-dimensional weighted balanced polyhedral fans, contained in two-dimensional matroidal fans trop(Y) where Y is a projective plane.

Notation: If Y is a projective plane in (n-1)-dimensional projective space, we consider trop(Y) in R^n/<1>. Moreover, for the relative realization space of C with respect to Y we only consider algebraic curves of degree deg(C) in Y which tropicalize to C.

Procedures:

`realizationDim(I,C)`
: For a given tropical fan curve C in trop(Y), where Y = V(I) is a projective plane, this routine returns the dimension of the relative realization space of C with respect to Y, that is the space of all algebraic curves of degree deg(C) in Y which tropicalize to C. If the realization space is empty, the output is set to -1.

`irrRealizationDim(I,C)`
: This routine returns the dimension of the irreducible relative realization space of the tropical fan curve C with respect to Y = V(I), that is the space of all irreducible algebraic curves of degree deg(C) in Y which tropicalize to C. If the irreducible relative realization space is empty, the output is set to -1.

`realizationDimPoly(I,C)`
If C is a tropical fan curve contained in the tropicalization trop(Y) of the projective plane Y = V(I) such that the relative realization space M of C is non-empty, this routine returns the tuple (dim(M),f) where f is an example of a homogeneous polynomial of degree deg(C) cutting out a curve X in Y which tropicalizes to C. If M is empty, the output is set to -1.

D.13.4 tropical_lib

Library: tropical.lib

Purpose: Computations in Tropical Geometry

Authors: Anders Jensen Needergard, email: jensen@math.tu-berlin.de
Hannah Markwig, email: hannah@uni-math.gwdg.de
Thomas Markwig, email: keilen@mathematik.uni-kl.de
Yue Ren, email: ren@mathematik.uni-kl.de

Warning: - tropicalLifting will only work with LINUX and if in addition gfan is installed.
- drawTropicalCurve and drawTropicalNewtonSubdivision will only display the tropical curve with LINUX and if in addition latex and xdg-open are installed.
- For tropicalLifting in the definition of the basering the parameter t from the Puiseux series field C{{t}} must be defined as a variable, while for all other procedures it must be defined as a parameter.

Theory: Fix some base field K and a bunch of lattice points v0,...,vm in the integer lattice Z^n, then this defines a toric variety as the closure of (K*)^n in the projective space P^m, where the torus is embedded via the map sending a point x in (K*)^n to the point (x^v0,...,x^vm).
The generic hyperplane sections are just the images of the hypersurfaces in (K*)^n defined by the polynomials f=a0*x^v0+...+am*x^vm=0. Some properties of these hypersurfaces can be studied via tropicalisation.

For this we suppose that K=C{{t}} is the field of Puiseux series over the field of complex numbers (or any other field with a valuation into the real numbers). One associates to the hypersurface given by f=a0*x^v0+...+am*x^vm the tropical hypersurface defined by the tropicalisation trop(f)=min{val(a0)+<v0,x>,...,val(am)+<vm,x>}. Here, <v,x> denotes the standard scalar product of the integer vector v in Z^n with the vector x=(x1,...,xn) of variables, so that trop(f) is a piecewise linear function on R^n. The corner locus of this function (i.e. the points at which the minimum is attained a least twice) is the tropical hypersurface defined by trop(f).
The theorem of Newton-Kapranov states that this tropical hypersurface is the same as if one computes pointwise the valuation of the hypersurface given by f. The analogue holds true if one replaces one equation f by an ideal I. A constructive proof of the theorem is given by an adapted version of the Newton-Puiseux algorithm. The hard part is to find a point in the variety over C{{t}} which corresponds to a given point in the tropical variety.

It is the purpose of this library to provide basic means to deal with tropical varieties. Of course we cannot represent the field of Puiseux series over C in its full strength, however, in order to compute interesting examples it will be sufficient to replace the complex numbers C by the rational numbers Q and to replace Puiseux series in t by rational functions in t, i.e. we replace C{{t}} by Q(t), or sometimes even by Q[t]. Note, that this in particular forbids rational exponents for the t's.

Moreover, in Singular no negative exponents of monomials are allowed, so that the integer vectors vi will have to have non-negative entries. Shifting all exponents by a fixed integer vector does not change the tropicalisation nor does it change the toric variety. Thus this does not cause any restriction.
If, however, for some reason you prefer to work with general vi, then you have to pass right away to the tropicalisation of the equations, whereever this is allowed – these are linear polynomials where the constant coefficient corresponds to the valuation of the original coefficient and where the non-constant coefficient correspond to the exponents of the monomials, thus they may be rational numbers respectively negative numbers: e.g. if f=t^{1/2}*x^{-2}*y^3+2t*x*y+4 then trop(f)=min{1/2-2x+3y,1+x+y,0}.

The main tools provided in this library are as follows:
- tropicalLifting implements the constructive proof of the Theorem of Newton-Kapranov and constructs a point in the variety over C{{t}} corresponding to a given point in the
corresponding tropical variety associated to an
ideal I; the generators of I have to be in the
polynomial ring Q[t,x1,...,xn] considered as a
subring of C{{t}}[x1,...,xn]; a solution will be
constructed up to given order; note that several
field extensions of Q might be necessary throughout
the intermediate computations; the procedures use
the external program gfan
- puiseuxExpansion computes a Newton-Puiseux expansion of a plane curve singularity
- drawTropicalCurve visualises a tropical plane curve either given by a polynomial in Q(t)[x,y] or by a list of linear
polynomials of the form ax+by+c with a,b in Z and c
in Q; latex must be installed on your computer
- tropicalJInvariant computes the tropical j-invaiant of a tropical elliptic curve
- jInvariant computes the j-invariant of an elliptic curve
- weierstrassForm computes the Weierstrass form of an elliptic curve

Procedures for tropical lifting:

`tropicalLifting()`
: computes a point in the tropical variety

`displayTropicalLifting()`
: displays the output of tropicalLifting

`puiseuxExpansion()`
: computes a Newton-Puiseux expansion in the plane

`displayPuiseuxExpansion()`
: displays the output of puiseuxExpansion

Procedures for drawing tropical curves:

`tropicalCurve()`
: computes a tropical curve and its Newton subdivision

`drawTropicalCurve()`
: produces a post script image of a tropical curve

`drawNewtonSubdivision()`

produces a post script image of a Newton subdivision PROCEDURES FOR J-INVARIANTS:

`tropicalJInvariant()`
computes the tropical j-invariant of a tropical curve

`weierstrassForm()`
computes the Weierstrass form of a cubic polynomial

`jInvariant()`
computes the j-invariant of a cubic polynomial

General procedures:

`conicWithTangents()`
computes a conic through five points with tangents

`tropicalise()`
computes the tropicalisation of a polynomial

`tropicaliseSet()`
computes the tropicalisation several polynomials

`tInitialForm()`
computes the tInitial form of a polynomial in Q[t,x_1,...,x_n]

`tInitialIdeal()`
computes the tInitial ideal of an ideal in Q[t,x_1,...,x_n]

`initialForm()`
computes the initial form of poly in Q[x_1,...,x_n]

`initialIdeal()`
computes the initial ideal of an ideal in Q[x_1,...,x_n]

Procedures for latex conversion:

`texNumber()`
outputs the texcommand for the leading coefficient of poly

`texPolynomial()`
outputs the texcommand for the polynomial poly

`texMatrix()`
outputs the texcommand for the matrix

`texDrawBasic()`
embeds output of texDrawTropical in a texdraw environment

`texDrawTropical()`
computes the texdraw commands for a tropical curve

`texDrawNewtonSubdivision()`
computes texdraw commands for a Newton subdivision

`texDrawTriangulation()`
computes texdraw commands for a triangulation

Auxilary procedures:

`radicalMemberShip()`
checks radical membership

`tInitialFormPar()`
: computes the t-initial form of poly in Q(t)[x_1,...,x_n]

`tInitialFormParMax()`
: same as tInitialFormPar, but uses maximum

`solveTInitialFormPar()`
: displays approximated solution of a 0-dim ideal

`detropicalise()`
: computes the detropicalisation of a linear form

`tDetropicalise()`
: computes the detropicalisation of a linear form

`dualConic()`
: computes the dual of an affine plane conic

`parameterSubstitute()`
: substitutes in the polynomial the parameter t by t^N

`tropicalSubst()`
: makes certain substitutions in a tropical polynomial

`randomPolyInT()`
: computes a polynomial with random coefficients

`cleanTmp()`
: clears /tmp from files created by other procedures

Procedures from binary library:

`groebnerCone()`
: constructs the Groebner cone with respect to a weight vector

`maximalGroebnerCone()`
: constructs the Groebner cone with respect to the current ordering

`homogeneitySpace()`
: constructs the homogeneity space

`initial()`
: constructs the initial form resp. ideal

`tropicalVariety()`
: computes the tropical variety of a poly or ideal

`groebnerFan()`
: computes the Groebner fan of a poly or ideal

`groebnerComplex()`
: computes the Groebner complex of a poly or ideal

D.14 Miscellaneous libraries

D.14.1 arr_lib

Library: arr.lib

Purpose: a library of algorithms for arrangements of hyperplanes

Authors: Randolf Scholz (rscholz@rhrk.uni-kl.de),
Patrick Serwene (serwene@mathematik.uni-kl.de),
Lukas Kuehne (lf.kuehne@gmail.com)

Overloads:

// OPERATORS
= arrAdd assignment
+ arrAdd union of two arrs
[arrGet access to a single/multiple hyperplane(s) - arrMinus deletes given hyperplanes from the arr <= arrLEQ comparison
>= arrGEQ comparison
== arrEQ comparison
!= arrNEQ comparison
< arrLNEQ comparison
> arrGNEQ comparison

// TYPECASTING
matrix arr2mat coeff matrix
poly arr2poly defining polynomial

// OTHER
variables arrVariables ideal generated by the variables the arr depends on nvars arrNvars number of variables the arr depends on delete arrDelete deletes hyperplanes by indices print arrPrint prints the arr on the screen

// IDEAL INHERITED FUNCTIONS
homog arrHomog checks if arrangement is homogeneous simplify arrSimplify simplifies arrangement size arrSize number of planes
subst arrSubst substitute variables

// MULTI-ARRANGEMENTS
= multarrAdd assignement of multarr + multarrAdd union of multarr
poly multarr2poly defining polynomial
size multarrSize number of hyperplanes with mult. print multarrPrint displays multiarr
delete multarrDelete deletes hyperplane

Procedures:

`arrSet(arr A, int k, poly p)`
replaces the k-th Hyperplane with poly p

`type2arr(#)`
converts general input to 'arr' using arrAdd.

`mat2arr( matrix M)`
affine arrangement from coeff matrix

`mat2carr(matrix M)`
central arrangement from coeff matrix

`arrPrintMatrix(arr A)`
readable output as a coeff matrix

`varMat(intvec v)`
: matrix of the corresponding ring_variables

`varNum(def u)`
: number of given variable (enh. version of varNum in dmod.lib)

`arrSwapVar(arr A, i, j)`
: swaps two variables in the arrangement

`arrLastVar(arr A)`
: ring_variable of largest index used in arrangement

`arrCenter(arr A)`
: computes center of an arrangement

`arrCentral(arr A)`
: checks if arrangement is central

`arrCentered(arr A)`
: checks if arrangement is centered

`arrCentralize(arr A)`
: makes centered arrangement central

`arrCoordChange(A, T, #)`
: performs coordinate change

`arrCoordNormalize(A, v)`
: performs projection onto coordinate hyperplane

`arrCone(arr A, var)`
: coned arrangement

`arrDecone(arr A, int k)`
: deconed arrangement

`arrLocalize(arr A, intvec v)`
: localization of an arrangement onto a flat

`arrRestrict(arr A, intvec v)`
: restricted arrangement onto a flat

`arrIsEssential(arr A)`
: checks if arrangement is essential

`arrEssentialize(arr A)`
: essentialized arragnement

`arrBoolean(int v)`
: boolean arrangement

`arrBraid(int v)`
: braid arrangement

`arrTypeB(int v)`
: type B arrangement

`arrTypeD(int v)`
: type D arrangement

`arrRandom(d,m,n)`
: random (affine) arrangement

`arrRandomCentral(d,m,n)`
: random central arrangement

`arrEdelmanReiner()`
: Edelman-Reiner arrangement

`arrOrlikSolomon(arr A)`
: Orlik-Solomon algebra of the arrangement

`arrDer(A)`
: module of derivation

`arrIsFree(A)`
: checks if arrangement is free

`arrExponents(A)`
: exponents of a (free) arrangement

`arr2multarr(arr A, intvec v)`
: converts normal arrangement to multiarrangement

`multarr2arr(multarr A)`
: converts multiarrangement to normal arrangement

`multarrRestrict(arr A, v)`
: restriction of A (as arr) to a flat with multiplicities

`multarrMultRestrict(A, int k)`
: restriction of A (as multarr) to a hyperplane with multiplicities

`arrFlats(arr A)`
: intersection lattice

`arrLattice(arr A)`
: computes the intersection lattice / poset

`moebius(arrposet P)`
: computes moebius values

`arrCharPoly(arr A)`
: characteristic polynomial

`arrPoincare(arr A)`
: poincare polynomial of the arrangement

`arrChambers(arr A)`
: number of chambers of the arrangement

`arrBoundedChambers(arr A)`
: number of bounded chambers of the arrangement

`printMoebius(arr A)`
: displays the moebius values of all the flats in the poset

D.14.2 customstd_lib

Library: customstd.lib

Purpose: Load customstd.so

Authors: Hans Schoenemann, hannes at mathematik.uni-kl.de
Yue Ren, ren at mathematik.uni-kl.de

Overview: This library offers customly modified standard bases algorithms in order to increase the performance of other algorithms. If you require a customly modified standard bases algorithm, please contact the authors.

Procedures:

`monomialabortstd(ideal)`

`satstd(ideal, [...])`

D.14.3 phindex_lib

Library : phindex.lib

Purpose: Procedures to compute the index of real analytic vector fields

Author: Victor Castellanos

Note: To compute the Poincare-Hopf index of a real analytic vector field with an algebraically isolated singularity at 0 (w. an a. i. s), we use the algebraic formula for the degree of the real analytic map germ found by Eisenbud-Levine in 1997. This result was also proved by Khimshiashvili. If the isolated singularity is non algebraically isolated and the vector field has similar reduced complex zeroes of codimension 1, we use a formula as the Eisenbud-Levine found by Victor Castellanos, in both cases is necessary to use a local order (ds,...). To compute the signature of a quadratic form (or symmetric matrix) we use the method of Lagrange.

Procedures:

`signatureL(M[,n])`
signature of symmetric matrix M, method of Lagrange.

`signatureLqf(h[,n])`
signature of quadratic form h, method of Lagrange.

`PH_ais(I)`
P-H index of real analytic vector field I w. an a. i. s.

`PH_nais(I)`
P-H index of real analytic vector field I w. a non a. i. s

D.14.4 polybori_lib

Library: polybori.lib

Purpose: A Singular Library Interface for PolyBoRi

Authors: Maximilian Kammermeier: Max0791@gmx.de
Susanne Scherer: sscherer90@yahoo.de

Overview: A library for using `PolyBoRi` in the SINGULAR interface, with procedures that convert structures (polynomials, rings, ideals) in both directions. Therefore, it is possible to compute boolean groebner basis via [boolean_std], page 825. Polynomials can be converted to zero-supressed decision diagrams (zdd) and vice versa.

For usability it defines the `PolyBoRi` types `bideal`, `bpoly`, and `bring` which are equivalent to Singular's `ideal`, `poly`, and `ring`, as well as `bset` which corresponds to the type `zdd` introduced here. In addition `bvar(i)` constructs the Boolean variable corresponding to `var(i)` from current `ring`;

For convenience, the corresponding types can be converted explictely or implicitely while assigning. Also several SINGULAR operators were overloaded: `bring` comes with `nvars`, `bpoly` implements `lead`, `leadmonom` and `leadcoef`. Objects of this type may be added and multiplied, too. Finally, `bideal` yields `std` and `size` as well as addition and element access.

Hence, by using these types `PolyBoRi` functionality can be carried out seamlessly in SINGULAR:

```
> LIB "polybori.lib";
> ring r0=2,x(1..4),lp;
> def x=bvar; // enforce Boolean variables

> bpoly f1=x(1)+x(4);
> bpoly f2=x(1)+x(3)*x(1);
> bideal bI=list(f1,f2);

> std(bI);
_[1] = x(1) + x(4)
_[2] = x(3)*x(4) + x(4)
```

Note: For using this library SINGULAR's `python` interface must be available on your system. Please `./configure --with-python` when building SINGULAR for this purpose.

There are prebuilt binary packages for `PolyBoRi` available from `http://polybori.sf.net/`.

For building your own `PolyBoRi` please ensure that you have `scons` and a development version of the boost libaries installed on you system. Then you may execute the following commands in a `bash`-style shell to build `PolyBoRi` available to `python`:

```
PBDIR=/path/to/custom/polybori
wget http://downloads.sf.net/project/polybori/polybori/\
0.8.2/polybori-0.8.2.tar.gz
tar -xvzf polybori-0.8.2.tar.gz
cd polybori-0.8.2
scons install PREFIX=$PBDIR PYINSTALLPREFIX=$PBDIR/python
export PYTHONPATH=$PBDIR/python:$PYTHONPATH
```

References:

See `http://polybori.sf.net` for details about `PolyBoRi`.

Procedures:

`boolean_std(bideal)`
Singular ideal of boolean groebner basis of I

`boolean_poly_ring(ring)`
convert ring

`boolean_constant(int[, bring])`
convert constant

`boolean_poly(poly[, int, bring])`
convert polynomial

`direct_boolean_poly(poly[, bring])`
convert polynomial direct

`recursive_boolean_poly(poly[, bring])`
convert polynomial recursively

`boolean_ideal(ideal[, bring])`
convert ideal

`boolean_set(zdd[, bring])`
convert zdd

`from_boolean_constant(bpoly)`
convert boolean constant

`from_boolean_poly(bpoly[, int])`
convert boolean polynomial

`direct_from_boolean_poly(bpoly)`
convert boolean polynomial direct

`recursive_from_boolean_poly(bpoly)`
convert boolean polynomial recursively

`from_boolean_ideal(bpoly)`
convert to ideal

`from_boolean_set(bset)`
convert to zdd

`bvar(i)` return i-th Boolean variable

`poly2zdd(poly)`
return zdd of a given polynomial

`zdd2poly(zdd)`
return polynomial representation of a given zdd

`disp_zdd(zdd)`
return string with a visualization of a given

See also: Section 3.8 [Libraries], page 55; Section 4.22 [User defined types], page 133; Section 4.26 [pyobject], page 139.

D.15 Experimental libraries

This sections collect libraries in the beta test phase. Everything in these libraries may change.

For the minimal requirements and guidelines see Section 3.8 [Libraries], page 55.

Comments should be send to the author of the library directly.

D.15.1 autgradalg_lib

Library: autgradalg.lib

Purpose: Compute automorphism groups of pointedly graded algebras and of Mori dream spaces.

Authors: Simon Keicher

Overview: This library provides a framework for computing automorphisms of integral, finitely generated algebras that are graded by a finitely generated abelian group. This library also contains functions to compute automorphism groups of Mori dream spaces. The results are ideals I such that the respective automorphism group is isomorphic to the subgroup V(I) in some GL(n).

Assumptions:
* the algebra R is given as factor algebra S/I with a graded polynomial ring S = KK[T_1,...,T_r]. We will always assume that the basering is S and it is given over the rationals QQ or a number field QQ(a). * R must be minimally presented, i.e., I is contained in <T_1,...,T_r>^2. * S (and hence R) are graded via 'setBaseMultigrading(Q)' from 'multigrading.lib'. The last rows of the matrix Q are interpreted over ZZ/a_iZZ if the respective entry of the list TOR is a_i and has been provided as parameter to the respective function. (See the functions for more details.) * For all 1 <= i <= r: I_{w_i} = 0 where w_i := deg(T_i). * the grading is pointed, i.e., no generator has degree 0 and the cone over all generator degrees is pointed. * For Mori dream spaces X, we assume them to be given as X = X(R,w) with the Cox ring R of X (given as the algebra R before) and an ample class w in the grading group K with the torsion entries removed.

Note: we require the user to execute 'LIB'gfanlib.so" before using this library.

Procedures:

`autKS()` : compute the automorphism group of the basering (must be a polynomial ring) as an algebraic subgroup V(I) of some GL(n)

`autGradAlg(I0, TOR)`
: compute the automorphism group of R as an algebraic subgroup V(I) of some GL(n).

`autGenWeights()`
: compute the automorphisms of the grading group that fix the generator degrees.

`stabilizer(I0, TOR)`
: compute the stabilizer of the given ideal

`autXhat(I0, w, TOR)`
: compute the automorphism group of widehat X as an algebraic subgroup V(I) of some GL(n).

`autX(I0, w, TOR)`
: compute the automorphism group of X=X(R,w) as an algebraic subgroup V(I) of some GL(n).

Note: the following functions were taken from 'quotsingcox.lib' by M.Donten-Bury and S.Keicher: 'hilbBas'.

D.15.2 brillnoether_lib

Library: brillnoether.lib

Purpose: Riemann-Roch spaces of divisors on curves

Authors: I. Stenger: stenger@mathematik.uni-kl.de
Janko Boehm boehm@mathematik.uni-kl.de

Procedures:

`RiemannRochBN()`
: Computes a vector space basis of the Riemann-Roch space of a divisor D on a projective curve C

D.15.3 chern_lib

Library: chern.lib

Purpose: Symbolic Computations with Chern classes, Computation of Chern classes

Author: Oleksandr Iena, o.g.yena@gmail.com, yena@mathematik.uni-kl.de

Overview: A toolbox for symbolic computations with Chern classes. The Aluffi's algorithms for computation of characteristic classes of algebraic varieties (Segre, Fulton, Chern-Schwartz-MacPherson classes) are implemented as well.

References:

[1] Aluffi, Paolo Computing characteristic classes of projective schemes. Journal of Symbolic Computation, 35 (2003), 3-19. [2] Iena, Oleksandr, On symbolic computations with Chern classes: remarks on the library chern.lib for Singular, http://hdl.handle.net/10993/22395, 2015.
[3] Lascoux, Alain, Classes de Chern d'un produit tensoriel. C. R. Acad. Sci., Paris, Ser. A 286, 385-387 (1978). [4] Manivel, Laurent Chern classes of tensor products, arXiv 1012.0014, 2010.

Procedures:

`symm(l [,N])`
: symmetric functions in the entries of l

`symNsym(f, c)`
: symmetric and non-symmetric parts of a polynomial f

`CompleteHomog(N, l)`
: complete homogeneous symmetric functions

`segre(N, c)`
: Segre classes in terms of Chern classes

`chern(N, s)`
: Chern classes in terms of Segre classes

`chNum(N, c)`
: the non-zero Chern numbers in degree N in the entries of c

`chNumbers(N, c)`
: the Chern numbers in degree N in the entries of c

`sum_of_powers(k, l)`
: the sum of k-th powers of the entries of l

`powSumSym(c [,N])`
: the sums of powers [up to degree N] in terms of the elementary symmetric polynomials (entries of l)

`chAll(c [,N])`
: Chern character in terms of the Chern classes

`chAllInv(c)`
: Chern classes in terms of the Chern character

`chHE(c)` the highest term of the Chern character

`ChernRootsSum(a, b)`
the Chern roots of a direct sum

`chSum(c, C)`
the Chern classes of a direct sum

`ChernRootsDual(l)`
the Chern roots of the dual vector bundle

`chDual(c)`
the Chern classes of the dual vector bundle

`ChernRootsProd(l, L)`
the Chern roots of a tensor product of vector bundles

`chProd(r, c, R, C [,N])`
Chern classes of a tensor product of vector bundles

`chProdE(c, C)`
Chern classes of a tensor product of vector bundles

`chProdL(r, c, R, C)`
Chern classes of a tensor product of vector bundles

`chProdLP(r, c, R, C)`
total Chern class of a tensor product of vector bundles

`chProdM(r, c, R, C)`
Chern classes of a tensor product of vector bundles

`chProdMP(r, c, R, C)`
total Chern class of a tensor product of vector bundles

`ChernRootsHom(l, L)`
the Chern roots of a Hom vector bundle

`chHom(r, c, R, C [,N])`
Chern classes of the Hom-vector bundle

`ChernRootsSymm(n, l)`
the Chern roots of the n-th symmetric power of a vector bundle with Chern roots from l

`ChernRootsWedge(n, l)`
the Chern roots of the n-th exterior power of a vector bundle with Chern roots from l

`chSymm(k, r, c [,p])`
the rank and the Chern classes of the k-th symmetric power of a vector bundle of rank r with Chern classes c

`chSymm2L(r, c)`
the rank and the Chern classes of the second symmetric power of a vector bundle of rank r with Chern classes c

`chSymm2LP(r, c)`
the total Chern class of the second symmetric power of a vector bundle of rank r with Chern classes c

`chWedge(k, r, c [,p])`
: the rank and the Chern classes of the k-th exterior power of a vector bundle of rank r with Chern classes c

`chWedge2L(r, c)`
: the rank and the Chern classes of the second exterior power of a vector bundle of rank r with Chern classes c

`chWedge2LP(r, c)`
: the total Chern class of the second exterior power of a vector bundle of rank r with Chern classes c

`todd(c [,n])`
: the Todd class

`toddE(c)` the highest term of the Todd class

`Bern(n)` the second Bernoulli numbers

`tdCf(n)` the coefficients of the Todd class of a line bundle

`tdTerms(n, f)`
: the terms of the Todd class of a line bundle coresponding to the Chern root t

`tdFactor(n, t)`
: the Todd class of a line bundle coresponding to the Chern root t

`cProj(n)` the total Chern class of (the tangent bundle on) the projective space P_n

`chProj(n)`
: the Chern character of (the tangent bundle on) the projective space P_n

`tdProj(n)`
: the Todd class of (the tangent bundle on) the projective space P_n

`eulerChProj(n, r, c)`
: Euler characteristic of a vector bundle on the projective space P_n via Hirzebruch-Riemann-Roch theorem

`chNumbersProj(n)`
: the Chern numbers of the projective space P_n

`classpoly(l, t)`
: polynomial in t with coefficients from l (without constant term)

`chernPoly(l, t)`
: Chern polynomial (constant term 1)

`chernCharPoly(r, l, t)`
: polynomial in t corresponding to the Chern character (constant term r)

`toddPoly(td, t)`
: polynomial in t corresponding to the Todd class (constant term 1)

`rHRR(N, ch, td)`
: the main ingredient of the right-hand side of the Hirzebruch-Riemann-Roch formula

`SchurS(I, S)`
: the Schur polynomial corresponding to partition I in terms of the Segre classes S

`SchurCh(I, C)`
: the Schur polynomial corresponding to partition I in terms of the Chern classes C

`part(m, n)`
: partitions of integers not exceeding n into m non-negative summands

`dualPart(I [,N])`
: partition dual to I

`PartC(I, m)`
: the complement of a partition with respect to m

`partOver(n, J)`
: partitions over a given partition J with summands not exceeding n

`partUnder(J)`
: partitions under a given partition J

`SegreA(I)`
: Segre class of the projective subscheme defined by I

`FultonA(I)`
: Fulton class of the projective subscheme defined by I

`CSMA(I)` Chern-Schwartz-MacPherson class of the projective subscheme defined by I

`EulerAff(I)`
: Euler characteristic of the affine subvariety defined by I

`EulerProj(I)`
: Euler characteristic of the projective subvariety defined by I

D.15.4 cimonom_lib

Library: cimonom.lib

Purpose: Determines if the toric ideal of an affine monomial curve is a complete intersection

Authors: I.Bermejo, ibermejo@ull.es
I.Garcia-Marco, iggarcia@ull.es
J.-J.Salazar-Gonzalez, jjsalaza@ull.es

Overview: A library for determining if the toric ideal of an affine monomial curve is a complete intersection with NO NEED of computing explicitly a system of generators of such ideal. It also contains procedures to obtain the minimum positive multiple of an integer which is in a semigroup of positive integers. The procedures are based on a paper by Isabel Bermejo, Ignacio Garcia and Juan Jose Salazar-Gonzalez: 'An algorithm to check whether the toric ideal of an affine monomial curve is a complete intersection', Preprint.

Procedures:

`BelongSemig(n,v[,sup])`
: checks whether n is in the semigroup generated by v;

`MinMult(a,b)`
: computes k, the minimum positive integer such that k*a is in the semigroup of positive integers generated by the elements in b.

`CompInt(d)`
checks wether I(d) is a complete intersection or not.

See also: Section C.6.4 [Integer programming], page 711.

D.15.5 classify2_lib

Library: classify2.lib

Purpose: Classification of isolated singularities

Authors: Janko Boehm, email: boehm@mathematik.uni-kl.de
Magdaleen Marais, email: magdaleen.marais@up.ac.za
Gerhard Pfister, email: pfister@mathematik.uni-kl.de

Overview: We classify isolated singularities of corank <=2 and modality <=2 with respect to right-equivalence over the complex numbers according to Arnold's list. We determine the type and, for positive modality, the parameter.

V.I. Arnold has described normal forms and has developed a classifier for, in particular, all isolated hypersurface singularities over the complex numbers up to modality 2. Building on a series of 105 theorems, this classifier determines the type of the given singularity. However, for positive modality, this does not fix the right equivalence class of the singularity, since the values of the moduli parameters are not specified.

This library implements an alternative classification algorithm for isolated hypersurface singularities of corank and modality up to two. For a singularity given by a polynomial over the rationals, the algorithm determines its right equivalence class by specifying a polynomial representative in Arnold's list of normal forms. In particular, the algorithm also determines values for the moduli parameters.

The implementation is based on the paper

Janko Boehm, Magdaleen Marais, Gerhard Pfister: A Classification Algorithm for Complex Singularities of Corank and Modality up to Two, Singularities and Computer Algebra - Festschrift for Gert-Martin Greuel on the Occasion of his 70th Birthday, Springer 2017, http://arxiv.org/abs/1604.04774, https://doi.org/10.1007/978-3-319-28829-1_2

There are functions for determining a normal form equation and for determining the complex type of the singularity.

Acknowledgements: This research was supported by
the Staff Exchange Bursary Programme of the University of Pretoria, DFG SPP 1489, DFG TRR 195. The financial assistance of the National Research Foundation (NRF), South Africa, towards this research is hereby acknowledged. Opinions expressed and conclusions arrived at are those of the author and are not necessarily to be attributed to the National Research Foundation, South Africa.

Procedures:

`complexClassify(I)`
classifier returning a normal form equation

`complexType(I)`
classifier returning the type and modality

See also: Section D.6.3 [classify_lib], page 774; Section D.15.31 [realclassify_lib], page 861.

D.15.6 classify_aeq_lib

Library: classifyAeq.lib

Purpose: Simple Space Curve singularities in characteristic 0

Authors: Faira Kanwal Janjua fairakanwaljanjua@gmail.com
Gerhard Pfister pfister@mathematik.uni-kl.de Khawar Mehmood khawar1073@gmail.com NEU

Overview: A library for classifying the simple singularities
with respect to A equivalence in characteristic 0.
Simple Surface singularities in characteristic O have been classified by Bruce and Gaffney [4] resp. Gibson and Hobbs [1] with respect to A equivalence. If the input is one of the simple singularities in [1] it returns a normal form otherwise a zero ideal(i.e not simple).

References:

[1] Gibson,C.G; Hobbs,C.A.:Simple SIngularities of Space Curves. Math.Proc. Comb.Phil.Soc.(1993),113,297.
[2] Hefez,A;Hernandes,M.E.:Standard bases for local rings of branches and their modules of differentials. Journal of Symbolic Computation 42(2007) 178-191. [3] Hefez,A;Hernandes,M.E.:The Analytic Classification Of Plane Branches. Bull.Lond Math Soc.43.(2011) 2,289-298. [4] Bruce, J.W.,Gaffney, T.J.: Simple singularities of mappings (C, 0) ->(C2,0). J. London Math. Soc. (2) 26 (1982), 465-474.
[5] Ishikawa,G; Janeczko,S.: The Complex Symplectic Moduli Spaces of Unimodal Parametric Plane Curve NEU Singularities. Insitute of Mathematics of the Polish Academy of Sciences,Preprint 664(2006)

Procedures:

`sagbiAlg(G)`
Compute the Sagbi-basis of the Algebra.

`sagbiMod(I,A)`
Compute the Sagbi- basis of the Module.

`semiGroup(G)`
Compute the Semi-Group of the Algebra provided the input is Sagbi Bases of the Algebra.

`semiMod(I,A)`
Compute the Semi-Module provided that the input are the Sagbi Bases of the Algebra resp.Module.

`planeCur(I)`
Compute the type of the Simple Plane Curve singularity.

`spaceCur(I)`
Compute the type of the simple Space Curve singularity.

`HHnormalForm(I)`
computes for the parametrization defined by I normal form, semi group, semi module of differentials, Zariski number and moduli

`modSagbiAlg(G)`
modular variant of sagbiAlg

D.15.7 classifyMapGerms_lib

Library: classifyMapGerms.lib

Authors: Gerhard Pfister, pfister@mathematik.uni-kl.de
Deeba Afzal, deebafzal@gmail.com
Shamsa Kanwal, lotus_zone16@yahoo.com

Overview: A library for computing the standard basis of the tangent space at the orbit of an algebraic group action. The tangent space is usually described as the sum of two modules over different rings. It computes the standard basis using modular methods and parallel modular methods. It also computes the normal form of the germ given by Riegers classification.

References:

[1] Idrees N.; Pfister, G.; Steidel, S.: Parallelization of modular algorithms. J. Symbolic Comput. 46(2011), no. 6, 672-684.
[2] Gibson,C.G; Hobbs,C.A.: Simple SIngularities of Space Curves. Math.Proc. Comb.Phil.Soc.(1993),113,297.
[3] Bruce, J.W.,Gaffney, T.J.: Simple singularities of mappings (C, 0) ->(C2,0). J. London Math. Soc. (2) 26 (1982), 465-474.
[4] Rieger, J. H.: Families of maps from the plane to the plane. J. London Math. Soc. (2)36(1987), no. 2. 351-369.

Procedures:

`coDimMap(I)`
computes a bound of the A-determinacy of the map germ defined by I

`coDim(M,N,I,b)`
computes the K-vectorspace dimension of A^r/M+N+maxideal(b)*A^r

`vStd(M,N,I,b)`
computes a standard basis of M+N+maxideal(b)*A^r

`modVStd(M,N,I,b)`
computes a standard basis of M+N+maxideal(bound)*A^r (modular)

`modVStd0(M,N,I,b)`
computes a standard basis of M+N+maxideal(bound)*A^r (parallel)

`classifySimpleMaps(I)`
computes the normal form of a germ in Riegers classification

`classifySimpleMaps1(I)`
computes the normal form of a germ in Riegers classification

`classifyUnimodalMaps(I)`
computes the normal form of a germ in Riegers classification

D.15.8 combinat_lib

Library: combinat.lib

Purpose: Some useful functions

Authors: J. Boehm, boehm @ mathematik.uni-kl.de

Overview: Some useful basic functions from combinatorics.

Procedures:

`intersectLists(list,list)`
: returns a list of elements which are in both lists (no duplicates)

`sublists(list)`
: all sublists

`member(def,list)`
: test whether an element is a member of a list

D.15.9 curveInv_lib

Library: curveInv.lib

Purpose: A library for computing invariants of curves

Author: Peter Chini, chini@rhrk.uni-kl.de

Overview: This library provides a collection of procedures for computing invariants of curve singularities. Invariants that can be computed are: - the delta invariant
- the multiplicity of the conductor: the length of Normalization(R)/C, where C denotes the conductor
- the Deligne number
- the colength of derivations along the normalization - the length of Der(Normalization(R/I)) / R/I

In addition, it is possible to compute the conductor of a ring S = R/I, where R is a (localized) polynomial ring.

Theory: Computing the Deligne number of curve singularities and an algorithmic framework for differential algebras in SINGULAR;
Chapter 5 - Master's Thesis of Peter Chini - August 2015

Procedures:

`curveDeltaInv(ideal)`
: computes the delta invariant of R/I for a given ideal I

`curveConductorMult(ideal)`
: returns the multiplicity of the conductor of R/I

`curveDeligneNumber(ideal)`
: computes the Deligne number of R/I

`curveColengthDerivations(ideal)`
: returns the colength of derivations, the length of Der(Normalization(R/I))/Der(R/I)

D.15.10 deRham_lib

Library: deRham.lib

Purpose: Computation of deRham cohomology

Authors: Cornelia Rottner, rottner@mathematik.uni-kl.de

Overview: A library for computing the de Rham cohomology of complements of complex affine varieties.

References:

[OT] Oaku, T.; Takayama, N.: Algorithms of D-modules - restriction, tensor product, localzation, and local cohomology groups}, J. Pure Appl. Algebra 156, 267-308 (2001)
[R] Rottner, C.: Computing de Rham Cohomology,diploma thesis (2012)
[W1] Walther, U.: Algorithmic computation of local cohomology modules and the local cohomological dimension of algebraic varieties}, J. Pure Appl. Algebra 139, 303-321 (1999)
[W2] Walther, U.: Algorithmic computation of de Rham Cohomology of Complements of Complex Affine Varieties}, J. Symbolic Computation 29, 796-839 (2000)
[W3] Walther, U.: Computing the cup product structure for complements of complex affine varieties, J. Pure Appl. Algebra 164, 247-273 (2001)

Procedures:

`deRhamCohomology(list[,opt])`
computes the de Rham cohomology

`MVComplex(list)`
computes the Mayer-Vietoris complex

D.15.11 divisors_lib

Library: divisors.lib

Purpose: Divisors and P-Divisors

Authors: Janko Boehm boehm@mathematik.uni-kl.de
Lars Kastner kastner@math.fu-berlin.de
Benjamin Lorenz blorenz@math.uni-frankfurt.de
Hans Schoenemann hannes@mathematik.uni-kl.de
Yue Ren ren@mathematik.uni-kl.de

Overview: We implement a class divisor on an algebraic variety and methods for computing with them. Divisors are represented by tuples of ideals defining the positive and the negative part. In particular, we implement the group structure on divisors, computing global sections and testing linear equivalence.

In addition to this we provide a class formaldivisor which implements integer formal sums of divisors (not necessarily prime). A formal divisor can be evaluated to a divisor, and a divisor can be decomposed into a formal sum.

Finally we provide a class pdivisor which implements polyhedral formal sums of divisors (P-divisors) where the coefficients are assumed to be polyhedra with fixed tail cone. There is a function to evaluate a P-divisor on a vector in the dual of the tail cone. The result will be a formal divisor.

References:

For the class divisor we closely follow Macaulay2's tutorial on divisors.

Procedures:

`makeDivisor(ideal,ideal)`
create a divisor

`divisorplus(divisor,divisor)`
add two divisors

`multdivisor(int,divisor)`
multiply a divisor by an interger

`negativedivisor(divisor)`
: compute the negative of the divisor

`normalForm(divisor)`
: normal form of a divisor

`isEqualDivisor(divisor,divisor)`
: test whether two divisors are equal

`globalSections(divisor)`
: compute the global sections of a divisor

`degreeDivisor(divisor)`
: degree of a divisor

`linearlyEquivalent(divisor,divisor)`
: test whether two divisors a linearly equivalent

`effective(divisor)`
: compute an effective divisor linearly equivalent to a divisor

`makeFormalDivisor(list)`
: make a formal integer sum of divisors

`evaluateFormalDivisor(formaldivisor)`
: evalutate a formal sum of divisors to a divisor

`formaldivisorplus(formaldivisor,formaldivisor)`
: add two formal divisors

`negativeformaldivisor(formaldivisor)`
: compute the negative of the formal divisor

`multformaldivisor(int,formaldivisor)`
: multiply a formal divisor by an interger

`degreeFormalDivisor(formaldivisor)`
: degree of a formal divisor

`makePDivisor(List)`
: make a formal polyhedral sum of divisors

`evaluatePDivisor(pdivisor,intvec)`
: evaluate a polyhedral divisor to an integer formal divisor

`pdivisorplus(pdivisor,pdivisor)`
: add two polyhedral divisors

D.15.12 ffmodstd_lib

Library: ffmodstd.lib

Purpose: Groebner bases of ideals in polynomial rings over rational function fields

Authors: D.K. Boku boku@mathematik.uni-kl.de
W. Decker decker@mathematik.uni-kl.de
C. Fieker fieker@mathematik.uni-kl.de

Overview: A library for computing a Groebner basis of an ideal in a polynomial ring over an algebraic function field Q(T):=Q(t_1,...,t_m) using modular methods and sparse multivariate rational interpolation, where the t_i are transcendental over Q. The idea is as follows: Given an ideal I in Q(T)[X], we map I to J via the map sending T to Tz:=(t_1z+s_1,..., t_mz+s_m) for a suitable point s in Q^m\{(0,...,0)} and for some extra variable z so that J is an ideal in Q(Tz)[X]. For a suitable point b in Z^m\{(0,...,0)}, we map J to K via the map sending (T,z) to (b,z), where b:=(b_1,...,b_m) (usually the b_i's are distinct primes), so that K is an ideal in Q(z)[X]. For such a rational point b, we compute a Groebner basis G_b of K using modular algorithms [1] and univariate rational interpolation [2,7]. The procedure is repeated for many rational points b until their number is sufficiently large to recover the correct coeffcients in Q(T). Once we have these points, we obtain a set of polynomials G by applying the sparse multivariate rational interpolation algorithm from [4] coefficient-wise to the list of Groebner bases G_b in Q(z)[X], where this algorithm makes use of the following algorithms: univariate polynomial interpolation [2], univariate rational function reconstruction [7], and multivariate polynomial interpolation [3]. The last algorithm uses the well-known Berlekamp/Massey algorithm [5] and its early termination version [6]. The set G is then a Groebner basis of I with high probability.

References:

[1] E. A. Arnold: Modular algorithms for computing Groebner bases. J. Symb. Comput. 35, 403-419 (2003).
[2] R. L. Burden and J. D. Faires: Numerical analysis. 9th ed. (1993).
[3] M. Ben-Or and P. Tiwari: A deterministic algorithm for sparse multivariate polynomial interpolation. Proc. of the 20th Annual ACM Symposium on Theory of Computing, 301-309 (1988).
[4] A. Cuyt and W.-s. Lee: Sparse interpolation of multivariate rational functions. Theor. Comput. Sci. 412, 1445-1456 (2011).
[5] E. Kaltofen and W.-s. Lee: Early termination in sparse interpolation algorithms. J. Symb. Comput. 36, 365-400 (2003).
[6] E. Kaltofen, W.-s. Lee and A. A. Lobo: Early termination in Ben-Or/Tiwari sparse interpolation and a hybrid of Zippel's algorithm. Proc. ISSAC (ISSAC '00), 192-201 (2000).
[7] K. Sara and M. Monagan: Fast Rational Function Reconstruction. Proc. ISSAC (ISSAC '06), 184-190 (2006).

Procedures:

`fareypoly(g,f)`
: univariate rational function reconstruction

`polyInterpolation(l,m)`
: univariate polynomial interpolation

`BerlekampMassey(L,i)`
: Berlekamp/Massey algorithm

`sparseInterpolation(f,L,n)`
: sparse multivariate polynomial interpolation

`ffmodStd(I)`
: Groebner bases over algebraic function fields using modular methods and sparse multivariate rational interpolation

D.15.13 finitediff_lib

Issues:

installation of qepcadfilter.pl needs to be solved

tests for (nearly) all procedures are missing
global variables needs to be cleaned
temporary files needs to be cleaned
temporary file names need to be unique (think about multiple instances)
pollution of global Top namespace must be solved
u is not a good name for a procedure

Library: finitediff.lib

Purpose: procedures to compute finite difference schemes for linear differential equations

Author: Christian Dingler

Overview: Using `qepcad/qepcadsystem` from this library requires the program `qepcad` to be installed. You can download `qepcad` from `http://www.usna.edu/CS/qepcadweb/B/QEPCAD.html`

Procedures:

`visualize(f)`
shows a scheme in index-notation

`u(D[,#])` gives some vector; depends on `@`derivatives

`scheme([v1,..,vn])`
computes the finite difference scheme defined by v1,..,vn

`laxfrT(Ut,U,space)`
Lax-Friedrich-approximation for the time-direction

`laxfrX(Ux,U,space)`
Lax-Friedrich-approximation for the space-direction

`forward(U1,U2,VAR)`
forward-approximation

`backward(U1,U2,VAR)`
backward-approximation

`central1st(U1,U2,VAR)`
central-approximation of first order

`central2nd(U1,U2,VAR)`
central-approximation of second order

`trapezoid(U1,U2,VAR)`
trapezoid-approximation

`midpoint(U1,U2,VAR)`
midpoint-approximation

`pyramid(U1,U2,VAR)`
pyramid-approximation

`setinitials(variable,der[,#])`
constructs and sets the basering for further computations

`errormap(f)`
performs the Fouriertransformation of a poly

`matrixsystem(M,A)`
: gives the scheme of a pde-system as one matrix

`timestep(M)`
: gives the several timelevels of a scheme derived from a pde-system

`fouriersystem(M,A)`
: performs the Fouriertransformation of a matrix scheme

`PartitionVar(f,n)`
: partitions a poly into the var(n)-part and the rest

`ComplexValue(f)`
: computes the complex value of f, var(1) being the imaginary unit

`VarToPar(f)`
: substitute var(i) by par(i)

`ParToVar(f)`
: substitute par(i) by var(i)

`qepcad(f)`
: ask QEPCAD for equivalent constraints to f<1

`qepcadsystem(l)`
: ask QEPCAD for equivalent constraints to all eigenvals of some matrices being <1

D.15.14 goettsche_lib

Library: goettsche.lib

Purpose: Drezet's formula for the Betti numbers of the moduli space of Kronecker modules, Goettsche's formula for the Betti numbers of the Hilbert scheme of points on a surface, Macdonald's formula for the symmetric product

Author: Oleksandr Iena, o.g.yena@gmail.com

References:

[1] Drezet, Jean-Marc Cohomologie des varie'te's de modules de hauter nulle. Mathematische Annalen: 281, 43-85, (1988).

[2] Goettsche, Lothar, The Betti numbers of the Hilbert scheme of ponts on a smooth projective surface.
Mathematische Annalen: 286, 193-208, (1990).

[3] Macdonald, I. G., The Poincare polynomial of a symmetric product, Mathematical proceedings of the Cambridge Philosophical Society: 58, 563 - 568, (1962).

Procedures:

`GoettscheF(z, t, n, b)`
: The Goettsche's formula up to n-th degree

`PPolyH(z, n, b)`
: Poincare Polynomial of the Hilbert scheme of n points on a surface

`BettiNumsH(n, b)`
: Betti numbers of the Hilbert scheme of n points on a surface

`MacdonaldF(z, t, n, b)`
: The Macdonald's formula up to n-th degree

`PPolyS(z, n, b)`
: Poincare Polynomial of the n-th symmetric power of a variety

`BettiNumsS(n, b)`
: Betti numbers of the n-th symmetric power of a variety

`PPolyN(t, q, m, n)`
: Poincare Polynomial of the moduli space of Kronecker modules N (q; m, n)

`BettiNumsN(q, m, n)`
: Betti numbers of the moduli space of Kronecker modules N (q; m, n)

D.15.15 graal_lib

Library: graal.lib

Purpose: localization at prime ideals and their associated graded rings

Author: Magdaleen Marais, magdaleen@aims.ac.za
Yue Ren, ren@mathematik.uni-kl.de

Overview: This library is on a computational treatment of localizations at prime ideals and their associated graded rings based on a work of Mora. Not only does it construct a ring isomorphic to the localization of an affine coordinate ring at a prime ideal,
the algorithms in this library aim to exploit the topology in the localization by computing first and foremost in the associated graded ring and lifting the result to the localization afterwards.
Features include a check for regularity and the resolution of ideals.

References:

Mora, Teo: La queste del Saint Gr_a(A_L): A computational approach to local algebra
Marais, Magdaleen and Ren, Yue: Mora's holy graal: Algorithms for computing in localizations at prime ideals

Procedures:

`graalMixed(ideal L[,int t])`
: contruct graalBearer

`dimensionOfLocalization(def L)`
: dimension of the localization A_L of A at L

`systemOfParametersOfLocalization(def L)`
: system of parameter of the localization A_L of A at L

`isLocalizationRegular(def L)`
: test if localization A_L of A at L is regular

`warkedPreimageStd(warkedModule wM)`
: std for warkedModule

`resolutionInLocalization(ideal I, def L)`
: the resolution of I*A_L

D.15.16 gitfan_lib

Library: gitfan.lib

Purpose: Compute GIT-fans.

Authors: Janko Boehm, boehm at mathematik.uni-kl.de
Simon Keicher, keicher at mail.mathematik.uni-tuebingen.de
Yue Ren, ren at mathematik.uni-kl.de

Overview: This library allows you to calculate GIT-fans, torus orbits and GKZ-fans.

In provides features to make use of symmetries of the torus action under consideration.

The main procedure is GITfan which can be directly applied to an ideal and a grading matrix encoding the torus action, and returns a fan, the associated GIT-fan. We also provide various procedures implementing substeps of the algorithm to deal with large computations.

The library uses the package 'gfanlib' by Anders N. Jensen.

For notation, background, and algorithms see [BKR16].

Functions produce debug output if printlevel is positive.

Elements of the symmetric group Sn of type permutation can be created by the function permutationFromIntvec.
The images of 1,...,n can be obtained by permutationToIntvec. Composition of permutations can be done by the *-Operator, also powers can be computed in the usual way.

References:

[BKR16] J. Boehm, S. Keicher, Y. Ren: Computing GIT-Fans with Symmetry and the Mori Chamber Decomposition of M06bar, https://arxiv.org/abs/1603.09241

Types: permutation; Permutation in map representation.

Procedures:

`isAface(ideal,intvec)`
: Checks whether the given face is an a-face.

`afaces(ideal)`
: Returns a list of intvecs that correspond to the set of all a-faces, optionally for given list of simplex faces.

`fullDimImages(list,intmat)`
: Finds the afaces which have a full-dimensional projection.

`minimalAfaces(list)`
: compute the minimal a-faces among the a-faces with full dimensional projection.

`orbitCones(list,intmat)`
: Returns the list of all orbit cones.

`GITcone(list,bigintmat)`
: Returns the GIT-cone containing the given weight vector.

`GITfan(ideal,intmat)`
: Compute GIT-fan.

`GITfanFromOrbitCones(list,intmat,cone)`
: Compute GIT-fan from orbit cones.

`GITfanParallel(list,intmat,cone)`
: Compute GIT-fan in parallel from orbit cones.

`GKZfan(intmat)`
: Returns the GKZ-fan of the matrix Q.

`movingCone(intmat)`
: Compute the moving cone.

`computeAfaceOrbits(list,list)`
: Compute orbits of a-faces under a permutation group action.

`minimalAfaceOrbits(list)`
: Find the minimal a-face orbits.

`orbitConeOrbits(list,intmat)`
: Project the a-face orbits to orbit cone orbits.

`minimalOrbitConeOrbits(list)`
: Find the minimal orbit cone orbits.

`intersectOrbitsWithMovingCone(list,cone)`
: Intersect orbit cone orbits with moving cone.

`groupActionOnQImage(list,intmat)`
: Determine the induced group action in the target of the grading matrix.

`groupActionOnHashes(list,list)`
: Determine the induced group action on the set of orbit cones.

`storeActionOnOrbitConeIndices(list, string)`
: Write the group action on the set of orbit cones to a file.

`permutationFromIntvec(intvec)`
: Create a permutation from an intvec of images.

`permutationToIntvec(permutation)`
: Return the intvec of images.

`evaluateProduct(list,list)`
: Evaluate a list of products of group elements in terms of a given representation of the elements as permutations.

`GITfanSymmetric(list,intmat,cone,list)`
: Compute GIT-fan from orbit cones by determining a minimal representing set for the orbits of maximal dimensional GIT-cones.

`GITfanParallelSymmetric(list, intmat, cone, list)`
: Compute GIT-fan in parallel from orbit cones by determining a minimal representing set for the orbits of maximal dimensional GIT-cones.

`bigintToBinary(bigint,int)`
: Convert bigint into a sparse binary represenation specifying the indices of the one-entries

`binaryToBigint(intvec)`
: Convert sparse binary represenation specifying the indices of the one-entries to bigint

`applyPermutationToIntvec(intvec,permutation)`
: Apply permutation to a set of integers represented as an intvec

`hashToCone(bigint,list)`
: Convert a bigint hash to a GIT-cone

`hashesToFan(list hashes,list OC)`

`gitCone(ideal,bigintmat,bigintmat)`
: Returns the GIT-cone around the given weight vector w

D.15.17 GND_lib

Library : GND.lib

Author : Adrian Popescu, popescu@mathematik.uni-kl.de

Overview :
A method to compute the General Neron Desingularization in the frame of one dimensional local domains

References:

[1] A. Popescu, D. Popescu, "A method to compute the General Neron Desingularization in the frame of one dimensional local domains", arxiv.org/abs/1508.05511

Procedures:

`desingularization()`

D.15.18 gradedModules_lib

Library: gradedModules.lib

Purpose: Operations with graded modules/matrices/resolutions

Authors: Oleksandr Motsak <U@D>, where U=motsak, D=mathematik.uni-kl.de
Hanieh Keneshlou <hkeneshlou@yahoo.com>

Overview: The library contains several procedures for constructing and manipulating graded modules/matrices/resolutions. Basics about graded objects can be found in [DL].
Throughout this library graded objects are graded maps, that is, matrices with polynomials, together with grading weights for source and destination. Graded modules are implicitly given as coker of a graded map. Note that in special cases we may also consider submodules in S^r generated by columns of a graded polynomial matrix (or a graded map).

Note: set assumeLevel to positive integer value in order to auto-check all assumptions. We denote the current basering by S.

References:
[DL] Decker, W., Lossen, Ch.: Computing in Algebraic Geometry, Springer, 2006

Procedures:

`grobj(M,w[,d])`
: construct a graded object (map) given by matrix M

`grtest(A)`
: check whether A is a valid graded object

`grdeg(M)` compute graded degrees of columns of the map M

`grview(M)`
view the graded structure of map M

`grshift(M,d)`
shift graded module coker(M) by +d

`grzero()` presentation of S(0)^1

`grtwist(r,d)`
presentation of S(d)^r

`grtwists(v)`
presentation of S(v[1])+...+S(v[size(v)])

`grsum(M,N)`
direct sum of two graded modules coker(M) + coker(N)

`grpower(M,p)`
direct p-th power of graded module coker(M)

`grtranspose(M)`
un-ordered graded transpose of map M

`grgens(M)`
try to compute submodule generators of coker(M)

`grpres(F)`
presentation of submodule generated by columns of F

`grorder(M)`
reorder cols/rows of M for correct graded-block-structure

`grtranspose1(M)`
reordered graded transpose of map M

`TestGRRes(n,I)`
compute/order/transpose a graded resolution of ideal I

`KeneshlouMatrixPresentation(v)`
build some presentation with intvec v

`grsyz(M)` syzygy of Im(M)

`grres(M,l[,m])`
resolution of Im(M) of length l... minimal?

`grlift(A,B)`
graded lift, gens!

`grprod(A,B)`
composition of graded maps (product of matrices?)

`grgroebner(M)`
Groebner Basis of Im(M) as a graded object

`grconcat(M,N)`
sum of maps into the same target module

`grrndmat(s,d[,p,b])`
generate random matrix compatible with src and dst gradings

`grrndmap(S,D[,p,b])`
: generate random 0-deg homomorphism src(S) -> src(D)

`grrndmap2(S,D[,p,b])`
: generate random 0-deg homomorphism dst(S) -> dst(D)

`grlifting(A,B)`
: RND! chain lifting

`grlifting2(A,B)`
: RND! chain lifting

`mappingcone(M,N)`
: mapping cone?

`grlifting3(A,B)`
: RND! chain lifting? probably wrong one

`mappingcone3(A,B)`
: mapping cone3?

`grrange(M)`
: get the row-weightings

`grneg(A)` graded object given by -A

`matrixpres(a)`
: matrix presentation of direct sum of Omega^{a[i]}(i)

D.15.19 hess_lib

Library: hess.lib

Purpose: Riemann-Roch space of divisors on function fields and curves

Authors: I. Stenger: stenger@mathematik.uni-kl.de

Overview: Let f be an absolutely irreducible polynomial in two variables x,y. Assume that f is monic as a polynomial in y. Let F = Quot(k[x,y]/f) be the function field defined by f. Define O_F = IntCl(k[x],F) and O_(F,inf) = IntCl(k[1/x],F). We represent a divisor D on F by two fractional ideals I and J of O_F and O_(F,inf), respectively. The Riemann-Roch space L(D) is then the intersection of I^(-1) and J^(-1).

Procedures:

`RiemannRochHess()`
: Computes a vector space basis of the Riemann-Roch space of a divisor

D.15.20 methods_lib

Library: methods.lib

Purpose: installing methods in Singular

Authors: J. Boehm, boehm @ mathematik.uni-kl.de

Overview: Methods select the function to execute by the types of the input tuple. The central function is installMethod, which takes a hashtable assocating a tuple of input types to function names and creates a corresponding procedure.

HashTables are lists with arbitrary index sets. They can be created by the command hashTable. Their size can be determined by the command size. Values can be extracted by selectKey or the * operator. HashTables can also be added using addHashTables or the + operator.

Methods can be added with the + operator.

Types: Method the class of all methods
HashTable the class of all hash tables

Procedures:

D.15.21 modules_lib

Library: modules.lib

Purpose: Modules

Authors: J. Boehm, boehm@mathematik.uni-kl.de
D. Wienholz wienholz@mathematik.uni-kl.de
C. Koenen koenen@rhrk.uni-kl.de
M. Mayer mayer@mathematik.uni-kl.de

Overview: This library is used for the computation of graded free resolutions with an own graduation of the monomials. For these Resolution is a new class of modules needed. These modules, can be computed via the image, kernel, cokernel of a matrix or the subquotient of two matrices. The used matrices also have a free module as source and target, with graded generators if the matrix is homogenous. A matrix of this new form is created by a normal matrix, source, target and the graduatin, if the matrix is homogenous, are done automatically. With this matrices it is then possible to compute the new class of modules.
This library also offers the opppurtunity to create R-module-homomorphisms betweens two modules. For these homorphisms the kernel can be computed an will be returned as a module of the new class.

This is experimental work in progress!!!

Types: Matrix the class of matrices with source and target in form of free modules FreeModule free modules representet with the ring and degree Resolution class of graded resolutions
Module modules represented by either the image, coker, kernelof a matrix or the subquotient of two matrices Vector element of a Module
Ideal same as ideal, but with it's own basering saved, used to compute resolutions Homomorphism class of R-module-homomormphisms

Procedures:

`id(int n)` return a nxn identity Matrix

`zero(int n,int m)`
return a nxm zero Matrix

`freeModule(ring,int,list)`
creating a graded free module

`makeMatrix(matrix,#int)`
creating a Matrix with graded target and source if the matrix is homogenous. If # is set to 1, makeMatrix ignores the grading of source & target.

`makeIdeal(ideal)`
: creates an Ideal from an given ideal, is used to compute a resolution of the ideal

`Target(Matrix)`
: return target of the Matrix

`Source(Matrix)`
: return source of the Matrix

`printMatrix(Matrix)`
: print a Matrix

`printFreeModule(FreeModule)`
: print a FreeModule

`printResolution(Resolution)`
: print a Resolution

`printModule(Module)`
: print a Module

`printHom(Homomorphism)`
: print a Homomorphism

`mRes(Module/Ideal,#int)`
: return a minimized graded Resolution

`sRes(Module/Ideal,#int)`
: return a graded Resolution computet with Schreyer's method

`Res(Module/Ideal,#int)`
: return a graded Resolution

`Betti(Resolution)`
: return the Betti-Matrix of the Resolution

`printBetti(Resolution)`
: prints the Betti-matrix of the Resolution

`SetDeg(list/intvec)`
: sets an own graduatuation for the monomials

`Deg(poly)`
: same as deg, but can be used with an own graduation

`Degree(FreeModule)`
: return list with degrees of the module

`Degrees(Module)`
: return list with degrees of the module

`subquotient(Matrix,Matrix)`
: return a Module, the subquotient of the two Matrices

`coker(Matrix)`
: return a Module, the cokernel of the Matrix

`image(Matrix)`
: return a Module, the image of the Matrix

`Ker(Matrix)`
: return a Module, the kernel of the Matrix

`compareModules(Module,Module)`
: return 0 or 1, compares the two Modules up to isomorphism

`addModules(Module,Module)`
: return a Module, sum of the two Modules

`homomorphism(matrix,Module,Module)`
: creates a R-Modul-Homomorphism

`target(Homomorphism)`
: return a Module, target of the Homomorphism

`source(Homomorphism)`
: return a Module, source of the Homomorphism

`compareMatrix(Matrix,Matrix)`
: return 0 or 1, compares two Matrices

`freeModule2Module(FreeModule)`
: converts a FreeModule into a Module

`makeVector(vector,Module)`
: creates Vector in the given Module

`netVector(Vector)`
: prints Vector

`netMatrix(Matrix)`
: prints Matrix

`presentation(Module)`
: converts M as a Subquotient to the Coker of a matrix C

`tensorMatrix(Matrix,Matrix)`
: computes tensorproduct of two Matrices

`tensorModule(Module,Module)`
: computes tensorproduct of two Modules

`tensorModFreemod(Module,FreeModule)`
: computes tensorproduct of Module and FreeModule

`tensorFreemodMod(FreeModule,Module)`
: computes tensorproduct of FreeModule and Module

`tensorFreeModule(FreeModule,FreeModule)`
: computes tensorproduct ot two FreeModules

`tensorProduct(def,def)`
: computes tensorproduct

`pruneModule(Module)`
: simplifies the presentation of a Module

`hom(Module,Module)`
: computes Hom(M,N)

`kerHom(Homomorphism)`
: computes the kernel of a Homomorphism

`interpret(Vector)`
: interpretes the Vector in some Module or abstract space

`interpretInv(def,Module)`
: interpretes a Vector or Homomorphism into the given Module

`reduceIntChain(Module,#int)`
: reduces a chain of interpretations to minimal size or # steps

`interpretElem(Vector,#int)`
: interpret a Vector with # steps or until can't interpret further

`interpretList(list,#int)`
: interpret a list of Vectors as far as possible

`compareVectors(Vector,Vector)`
: compares two Vectors with regard to the relations of their Module

`simplePrune(Module)`
: simplify module

D.15.22 modwalk_lib

Library: modwalk.lib

Purpose: Groebner basis convertion

Authors: S. Oberfranz oberfran@mathematik.uni-kl.de

Overview: A library for converting Groebner bases of an ideal in the polynomial ring over the rational numbers using modular methods. The procedures are inspired by the following paper:
Elizabeth A. Arnold: Modular algorithms for computing Groebner bases. Journal of Symbolic Computation 35, 403-419 (2003).

Procedures:

`modWalk(I,#)`
: standard basis conversion of I by Groebner Walk using modular methods

`modrWalk(I,radius,#)`
: standard basis conversion of I by Random Walk using modular methods

`modfWalk(I,#)`
: standard basis conversion of I by Fractal Walk using modular methods

`modfrWalk(I,radius,#)`
: standard basis conversion of I by Random Fractal Walk using modular methods

See also: Section D.4.7 [grwalk_lib], page 745; Section D.15.34 [rwalk_lib], page 864; Section D.15.36 [swalk_lib], page 866.

D.15.23 multigrading_lib

Todos/Issues:

See `http://code.google.com/p/convex-singular/wiki/Multigrading`

Library: multigrading.lib

Purpose: Multigraded Rings

Authors: Benjamin Bechtold, benjamin.bechtold@googlemail.com
Rene Birkner, rbirkner@math.fu-berlin.de
Lars Kastner, lkastner@math.fu-berlin.de
Simon Keicher, keicher@mail.mathematik.uni-tuebingen.de
Oleksandr Motsak, U@D, where U={motsak}, D={mathematik.uni-kl.de}
Anna-Lena Winz, anna-lena.winz@math.fu-berlin.de

Overview: This library allows one to virtually add multigradings to Singular: grade multivariate polynomial rings with arbitrary (fin. gen. Abelian) groups. For more see http://code.google.com/p/convex-singular/wiki/Multigrading For theoretical references see:
E. Miller, B. Sturmfels: 'Combinatorial Commutative Algebra' and
M. Kreuzer, L. Robbiano: 'Computational Commutative Algebra'.

Note: 'multiDegBasis' relies on 4ti2 for computing Hilbert Bases. All groups are finitely generated Abelian

Procedures:

`setBaseMultigrading(M,L)`
: attach multiweights/grading group matrices to the basering

`getVariableWeights([R])`
: get matrix of multidegrees of vars attached to a ring

`getGradingGroup([R])`
: get grading group attached to a ring

`getLattice([R[,choice]])`
: get grading group' lattice attached to a ring (or its NF)

`createGroup(S,L)`
: create a group generated by S, with relations L

`createQuotientGroup(L)`
: create a group generated by the unit matrix whith relations L

`createTorsionFreeGroup(S)`
: create a group generated by S which is torsionfree

`printGroup(G)`
: print a group

`isGroup(G)`
: test whether G is a valid group

`isGroupHomomorphism(L1,L2,A)`
: test wheter A defines a group homomrphism from L1 to L2

`isGradedRingHomomorphism(R,f,A)`
: test graded ring homomorph

`createGradedRingHomomorphism(R,f,A)`
: create a graded ring homomorph

`setModuleGrading(M,v)`
: attach multiweights of units to a module and return it

`getModuleGrading(M)`
: get multiweights of module units (attached to M)

`isSublattice(A,B)`
: test whether A is a sublattice of B

`imageLattice(P,L)`
: computes an integral basis for P(L)

`intRank(A)`
: computes the rank of the intmat A

`kernelLattice(P)`
: computes an integral basis for the kernel of the linear map P.

`latticeBasis(B)`
: computes an integral basis of the lattice B

`preimageLattice(P,L)`
: computes an integral basis for the preimage of the lattice L under the linear map P.

`projectLattice(B)`
: computes a linear map of lattices having the primitive span of B as its kernel.

`intersectLattices(A,B)`
: computes an integral basis for the intersection of the lattices A and B.

`isIntegralSurjective(P)`
: test whether the map P of lattices is surjective.

`isPrimitiveSublattice(A)`
: test whether A generates a primitive sublattice.

`intInverse(A)`
: computes the integral inverse matrix of the intmat A

`integralSection(P)`
: for a given linear surjective map P of lattices this procedure returns an integral section of P.

`primitiveSpan(A)`
: computes a basis for the minimal primitive sublattice that contains the given vectors (by A).

`factorgroup(G,H)`
: create the group G mod H

`productgroup(G,H)`
: create the group G x H

`multiDeg(A)`
: compute the multidegree of A

`multiDegBasis(d)`
: compute all monomials of multidegree d

`multiDegPartition(p)`
: compute the multigraded-homogeneous components of p

`isTorsionFree()`
: test whether the current multigrading is free

`isPositive()`
: test whether the current multigrading is positive

`isZeroElement(p)`
: test whether p has zero multidegree

`areZeroElements(M)`
: test whether an integer matrix M considered as a collection of columns has zero multidegree

`isHomogeneous(a)`
: test whether 'a' is multigraded-homogeneous

`equalMultiDeg(e1,e2[,V])`
: test whether e1==e2 in the current multigrading

`multiDegGroebner(M)`
: compute the multigraded GB/SB of M

`multiDegSyzygy(M)`
: compute the multigraded syzygies of M

`multiDegModulo(I,J)`
: compute the multigraded 'modulo' module of I and J

`multiDegResolution(M,l[,m])`
: compute the multigraded resolution of M

`multiDegTensor(m,n)`
: compute the tensor product of multigraded modules m,n

`multiDegTor(i,m,n)`
: compute the Tor_i(m,n) for multigraded modules m,n

`defineHomogeneous(p)`
: get a grading group wrt which p becomes homogeneous

`pushForward(f)`
: find the finest grading on the image ring, homogenizing f

`gradiator(h)`
: coarsens grading of the ring until h becomes homogeneous

`hermiteNormalForm(A)`
: compute the Hermite Normal Form of a matrix

`smithNormalForm(A,#)`
: compute matrices D,P,Q with D=P*A*Q and D is the smith normal form of A

`hilbertSeries(M)`
: compute the multigraded Hilbert Series of M

`lll(A)`
: applies LLL(.) of lll.lib which only works for lists on a matrix A

D.15.24 ncHilb_lib

Library: ncHilb.lib

Purpose: : A library for computing graded and multi-graded Hilbert series of non-commutative algebras. It also computes the truncated Hilbert series of the algebras whose Hilbert series are possibly not rational or unknown.

Author: Sharwan K. Tiwari shrawant@gmail.com
Roberto La Scala

References:

La Scala R.: Monomial right ideals and the Hilbert series of non-commutative modules, J. of Symb. Comp. (2016).

La Scala R., Tiwari Sharwan K.: Multigraded Hilbert Series of noncommutative modules, https://arxiv.org/abs/1705.01083.

Procedures:

`nchilb(L,d,#)`
Hilbert series of a non-commutative algebra

`rcolon(L, w, d)`
Right colon ideal of a two-sided monomial ideal with respect to a monomial w

D.15.25 ncfrac_lib

Library: ncfrac.lib

Purpose: object-oriented interface for olga.lib

Author: Johannes Hoffmann, email: johannes.hoffmann at math.rwth-aachen.de

Overview: This library introduces a new type: ncfrac.
This type wraps the data defining a (non-commutative) fraction in an Ore localization of a G-algebra as in olga.lib.
An element of type ncfrac has five members:
- polys lnum, lden, rnum, rden
- ncloc loc

Operations:

string(ncfrac);
give a string representation of the data describing the fraction print(ncfrac);
prints the string representation of the fraction
status(ncfrac);
report on the status/validity of the fraction
test(ncfrac);
check if the fraction is valid

Infix operations:

ncfrac == ncfrac;
compare two fractions
ncfrac != ncfrac;
compare two fractions
ncfrac + ncfrac;
add two fractions

ncfrac - ncfrac
subtract two fractions
ncfrac * ncfrac
multiply two fractions
ncfrac / ncfrac
divide two fractions
ncfrac = int/number/poly
create a fraction with:
- left and right denominator equal to 1
- left and right numerator determined by the input - localization data describing the trivial monoidal localization at 1 ncfrac = vector
create a fraction from a vector v with unspecified localization such that lden,lnum,rnum,rden = v[1],v[2],v[3],v[4]
(note: without specifying a localization afterwards this results is an invalid fraction)
ncfrac = list
create a fraction from a list L as follows: - try to create a fraction from L[1] as above
- if L[2] is of type ncloc set the localization of the fraction to L[2]

Procedures:

`hasLeftDenom(ncfrac)`
: check if the given fraction has a left representation

`hasRightDenom(ncfrac)`
: check if the given fraction has a right representation

`isZeroNcfrac(ncfrac)`
: check if the given fraction is equal to zero

`isOneNcfrac(ncfrac)`
: check if the given fraction is equal to one

`zeroNcfrac(ncloc loc)`
: create the fraction equal to zero in the given localization

`oneNcfrac(ncloc loc)`
: create the fraction equal to one in the given localization

`ensureLeftNcfrac(ncfrac)`
: compute a left representation of the given fraction if it does not have one

`ensureRightNcfrac(ncfrac)`
: compute a right representation of the given fraction if it does not have one

`negateNcfrac(ncfrac)`
: compute the additive inverse of the given fraction

`isInvertibleNcfrac(ncfrac)`
: check if the given fraction is invertible (note: see the description of isInvertibleLeftFraction from olga.lib for specific behaviour)

`invertNcfrac(ncfrac)`
: compute the multiplicative inverse of the given fraction (note: see the description of invertLeftFraction from olga.lib for specific behaviour)

`testNcfrac()`
: execute a series of internal testing procedures

`testNcfracExamples()`
: execute the examples of all procedures in this library

D.15.26 ncloc_lib

Library: ncloc.lib

Purpose: Ore-localization in G-Algebras

Author: Johannes Hoffmann, email: johannes.hoffmann at math.rwth-aachen.de

Overview: This library introduces a new type: ncloc.
This type wraps the localization data defined as in olga.lib. An element of type ncloc has two members:
- int locType
- def locData

Operations:
string(ncloc);
give a string representation of the data describing the localization print(ncloc);
prints the string representation of the localization status(ncloc);
report on the status/validity of the localization test(ncloc);
check if the localization is valid

Infix operations:
ncloc == ncloc;
compare two localizations
ncloc != ncloc;
compare two localizations
ncloc = list/poly
create a monoidal localization from the given data ncloc = ideal
create a geometric localization from the given data ncloc = intvec
create a rational localization from the given data

Procedures:

`isDenom(poly, ncloc)`
determine if a polynomial is in the denominator set of the localization

`testNcloc()`
execute a series of internal testing procedures

`testNclocExamples()`
execute the examples of all procedures in this library

D.15.27 nets_lib

Library: net.lib

Purpose: Net structures for pretty printing

Authors: J. Boehm, boehm@mathematik.uni-kl.de
M. Mueller, mkmuelle@mathematik.uni-kl.de
H. Rombach, rombach@mathematik.uni-kl.de
M. Stein, maxstein77@web.de

Overview: Nets are arrays of characters, which are printed in a matrix format. They can be concatenated horizontally and vertically. When concatenating horizontally, empty rows are filled with spaces. All Singular types can be converted to a Net by applying the command net.

Types: Net The class of all nets

Procedures:

`catNets(Net,Net)`
: horizontal concatenation

`net(def)` general procedure to generate a net from a Singular object

`netBigIntMat(bigintmat)`
: procedure to generate a net from a bigintmat

`netBigIntMatShort(bigintmat)`
: procedure to generate a net from a bigintmat

`netCoefficientRing(ring)`
: procedure to generate a net from a Singular coefficient ring

`netIdeal(ideal)`
: procedure to generate a net from an ideal

`netInt(int)`
: procedure to generate a net from a integer

`netBigInt(bigint)`
: procedure to generate a net from a bigint

`netIntMat(intmat)`
: procedure to generate a net from an intmat

`netIntMatShort(intmat)`
: procedure to generate a net from an intmat

`netIntVector(vector)`
: procedure to generate a net from an intvec

`netIntVectorShort(vector)`
: procedure to generate a net from an intvec

`netNumber(number)`
: procedure to generate a net from a number

`netList(list)`
: procedure to generate a net from a list

`netMap(map)`
: procedure to generate a net from a map

`netMap2(map)`
: procedure to generate a net from a map

`netmatrix(matrix)`
: procedure to generate a net from a matrix

`netmatrixShort(matrix)`
: procedure to generate a net from a matrix

`netPoly(poly)`
: procedure to generate a net from a poly

`netPrimePower(int,int)`
: procedure to generate a net from a prime power

`netRing(ring)`
procedure to generate a net from a polynmial ring

`netString(string)`
procedure to generate a net from a string

`netvector(vector)`
procedure to generate a net from a vector

`netvectorShort(vector)`
procedure to generate a net from a vector

`stackNets(Net,Net)`
vertical concatenation

D.15.28 nfmodstd_lib

Library: nfmodstd.lib

Purpose: Groebner bases of ideals in polynomial rings over algebraic number fields

Authors: D.K. Boku boku@mathematik.uni-kl.de
W. Decker decker@mathematik.uni-kl.de
C. Fieker fieker@mathematik.uni-kl.de

Overview: A library for computing the Groebner basis of an ideal in the polynomial ring over an algebraic number field Q(t) using the modular methods, where t is algebraic over the field of rational numbers Q. For the case Q(t) = Q, the procedure is inspired by Arnold [1]. This idea is then extended to the case t not in Q using factorization as follows:
Let f be the minimal polynomial of t.
For I, I' ideals in Q(t)[X], Q[X,t]/<f> respectively, we map I to I' via the map sending t to t + <f>. We first choose a prime p such that f has at least two factors in characteristic p and add each factor f_i to I' to obtain the ideal J'_i = I' + <f_i>. We then compute a standard basis G'_i of J'_i for each i and combine the G'_i to G_p (a standard basis of I'_p) using chinese remaindering for polynomials. The procedure is repeated for many primes p, where we compute the G_p in parallel until the number of primes is sufficiently large to recover the correct standard basis G' of I'. Finally, by mapping G' back to Q(t)[X], a standard basis G of I is obtained.
The procedure also works if the input is a module. For this, we consider the rings A = Q(t)[X] and A' = (Q[t]/<f>)[X]. For submodules I, I' in A^m, A'^m, respectively, we map I to I' via the map sending t to t + <f>. As above, we first choose a prime p such that f has at least two factors in characteristic p. For each factor f_{i,p} of f_p := (f mod p), we set I'_{i,p} := (I'_p mod f_{i,p}). We then compute a standard basis G'_i of I'_{i,p} over F_p[t]/<f_{i,p}> for each i and combine the G'_i to G_p (a standard basis of I'_p) using chinese remaindering for polynomials. The procedure is repeated for many primes p as described above and we finally obtain a standard basis of I.

References:
[1] E. A. Arnold: Modular algorithms for computing Groebner bases. J. Symb. Comp. 35, 403-419 (2003).

Procedures:

`chinrempoly(l,m)`
chinese remaindering for polynomials

`nfmodStd(I)`
standard basis of I over algebraic number field using modular methods

D.15.29 nfmodsyz_lib

Library: nfmodsyz.lib

Purpose: Syzygy modules of submodules of free modules over algebraic number fields

Authors: D.K. Boku boku@mathematik.uni-kl.de
W. Decker decker@mathematik.uni-kl.de
C. Fieker fieker@mathematik.uni-kl.de

Overview: A library for computing the syzygy module of a given submodule I in a polynomial ring over an algebraic number field Q(t), where t is an algebraic number, using modular methods. For the case Q(t)=Q, that is, where t is an element of Q, we compute, following [1], the syzygy module of I as follows: For a submodule I of A^m with A = Q[X], we first choose a sufficiently large set of primes P and compute the reduced Groebner basis of the syzygy module of I_p, for each p in P, in parallel. We then use the Chinese remainder algorithm and rational reconstruction to obtain the syzygy module of I over Q. For the case where t is not in Q, we compute, following [2], the syzygy module of I as follows:
Let f be the minimal polynomial of t. For a submodule I in A^m with A = Q(t)[X], we map I to a submodule I' in A^m with A = (Q[t]/<f>)[X] via the map sending t to t + <f>. We first choose a prime p such that f has at least two factors in characteristic p. For each factor f_{i,p} of f_p:= (f mod p), we set I'_{i,p} := (I'_p mod f_{i,p}). We then compute the reduced Groebner bases G'_i of the syzygy modules of I'_{i,p} over F_p[t]/<f_{i,p}> and combine the G'_i to G_p (the syzygy module of I'_p) using chinese remaindering for polynomials. As described in [2], the procedure is repeated for many primes p, where we compute the G_p in parallel until the number of primes is sufficiently large to recover the correct generating set for the syzygy module G' of I' which is, considered over Q(t), also a generating set for the syzygy module of I.

References:

[1] E. A. Arnold: Modular algorithms for computing Groebner bases. J. Symb. Comp. 35, 403-419 (2003).
[2] D. Boku, W. Decker, C. Fieker, and A. Steenpass. Groebner bases over algebraic number fields. In: Proceedings of the 2015 International Workshop on Parallel Symb. Comp. PASCO'15, pages 16-24 (2015).

Procedures:

`nfmodSyz(I)`
syzygy module of I over algebraic number field using modular methods

D.15.30 olga_lib

Library: olga.lib

Purpose: Ore-localization in G-Algebras

Author: Johannes Hoffmann, email: johannes.hoffmann at math.rwth-aachen.de

Overview: Let A be a G-algebra.

Current localization types:
Type 0: monoidal
- represented by a list of polys g_1,...,g_k that have to be contained in a commutative polynomial subring of A generated by a subset of the variables of A

Type 1: geometric
- only for algebras with an even number of variables where the first half induces a commutative polynomial subring B of A
- represented by an ideal p, which has to be a prime ideal in B Type 2: rational
- represented by an intvec v = [i_1,...,i_k] in the range 1..nvars(basering)

Localization data is an int specifying the type and a def with the corresponding information.

A fraction is represented as a vector with four entries: [s,r,p,t] Here, s^{-1}r is the left fraction representation, pt^{-1} is the right one. If s or t is zero, it means that the corresponding representation is not set. If both are zero, the fraction is not valid.

A detailed description along with further examples can be found in our paper: Johannes Hoffmann, Viktor Levandovskyy:
Constructive Arithmetics in Ore Localizations of Domains https://arxiv.org/abs/1712.01773

Procedures:

`locStatus(int, def)`
: report on the status/validity of the given localization data

`testLocData(int, def)`
: check if the given data specifies a denominator set

`isInS(poly, int, def)`
: determine if a polynomial is in a denominator set

`fracStatus(vector frac, int locType, def locData)`
: report on the status/validity of the given fraction wrt. to the given localization data

`testFraction(vector, int, def)`
: check if the given vector is a representation of a fraction in the specified localization

`leftOre(poly, poly, int, def)`
: compute left Ore data

`rightOre(poly, poly, int, def)`
: compute right Ore data

`convertRightToLeftFraction(vector, int, def)`
: determine a left fraction representation of a given fraction

`convertLeftToRightFraction(vector, int, def)`
: determine a right fraction representation of a given fraction

`addLeftFractions(vector, vector, int, def)`
: add two left fractions in the specified localization

`multiplyLeftFractions(vector, vector, int, def)`
: multiply two left fractions in the specified localization

`areEqualLeftFractions(vector, vector, int, def)`
: check if two given fractions are equal

`isInvertibleLeftFraction(vector, int, def)`
: check if a fraction is invertible in the specified localization (NOTE: check description for specific behaviour)

`invertLeftFraction(vector, int, def)`
: invert a fraction in the specified localization (NOTE: check description for specific behaviour)

`isZeroFraction(vector)`
: determine if the given fraction is equal to zero

`isOneFraction(vector)`
: determine if the given fraction is equal to one

`normalizeMonoidal(list)`
: determine a normal form for monoidal localization data

`normalizeRational(intvec)`
: determine a normal form for rational localization data

`testOlga()`
: execute a series of internal testing procedures

`testOlgaExamples()`
: execute the examples of all procedures in this library

D.15.31 realclassify_lib

Library: realclassify.lib

Purpose: Classification of real singularities

Author: Janko Boehm, boehm@mathematik.uni-kl.de
Magdaleen Marais, magdaleen@aims.ac.za
Andreas Steenpass, steenpass@mathematik.uni-kl.de

Overview: A library for classifying isolated hypersurface singularities over the reals w.r.t. right equivalence, based on the determinator of singularities by V.I. Arnold. This library is based on classify2.lib by the first and second author and G. Pfister, but handles the real case, while classify2.lib does the complex classification.

References:

Arnold, Varchenko, Gusein-Zade: Singularities of Differentiable Maps. Vol. 1: The classification of critical points caustics and wave fronts. Birkh"auser, Boston 1985

Greuel, Lossen, Shustin: Introduction to singularities and deformations. Springer, Berlin 2007

Procedures:

`realclassify(f)`
: real classification of singularities of modality 0 and 1 up to stable equivalence

`realmorsesplit(f)`
: splitting lemma in the real case

`milnornumber(f)`
: Milnor number

`determinacy(f)`
: an upper bound for the determinacy

`addnondegeneratevariables(f)`
: find a right equivalent normal form by adding the non-degenerate variables

D.15.32 rootisolation_lib

Library: rootisolation.lib

Purpose: implements an algorithm for real root isolation using interval arithmetic

Authors: Dominik Bendle (bendle@rhrk.uni-kl.de)
Janko Boehm (boehm@mathematik.uni-lk.de), supervisor Fachpraktikum
Clara Petroll (petroll@rhrk.uni-kl.de)

Overview: In this library the interval arithmetic from `interval.so` is used. The new type `ivmat`, a matrix consisting of intervals, is implemented as `newstruct`. There are various functions for computations with interval matrices implemented, such as Gaussian elimination for interval matrices.

Interval arithmetic, the interval Newton Step and exclusion methods are used to implement the procedure `rootIsolation`, an algorithm which finds boxes containing elements of the vanishing locus of an ideal. This algorithm is specialised for zero-dimensional radical ideals. The theory about the interval Newton Step is detailed in [2].

Note that interval arithmetic and the aforementioned procedures are intended for rational or real polynomial rings.

References:

[1] Cloud, Kearfott, Moore: Introduction to Interval Analysis, Society for Industrial and Applied Mathematics, 2009
[2] Eisenbud, Grayson, Herzog, Stillman, Vasconcelos: Computational Methods in Commutative Algebra and Algebraic Geometry, Springer Verlag Berlin-Heidelberg, 3. edition 2004
[3] Andrew J. Sommese and Charles W. Wampler: The Numerical Solution of Systems of Polynomials - Arising in Engineering and Science, World Scientific Publishing Co. Pte. Ltd., 2005

Overloads:

[ivmatGet indexing
print ivmatPrint printing
nrows ivmatNrows number of rows
ncols ivmatNcols number of columns
* ivmatMultiplyGeneral matrix multiplication

Procedures:

`bounds(a,#)`
    creates a new interval with given bounds

`length(I)`
    returns Euclidean length of interval

`boxSet(B,i,I)`
    returns box B with B[i]==I

`ivmatInit(m, n)`
    returns m x n [0,0]-matrix

`ivmatSet(A,i,j,I)`
    returns matrix A where A[i][j]=I

`unitMatrix(m)`
returns m x m unit matrix where 1 = [1,1]

`ivmatGaussian(M)`
computes M^(-1) using Gaussian elimination for intervals

`evalPolyAtBox(f,B)`
returns evaluation of polynomial at a box

`evalJacobianAtBox(A,B)`
jacobian matrix of A where polynomials are evaluated at B

`rootIsolationNoPreprocessing(I,L,e)`
computes boxes containing unique roots of I lying in L

`rootIsolation(I,B,e)`
slims down input box B and calls rootIsolationNoPreprocessing

`rootIsolationPrimdec(I,B,e)`
runs a primary decomposition primdecGTZ before root isoation

D.15.33 rstandard_lib

Library: rstandard.lib

Purpose: Computes Janet bases and border bases for ideals

Authors: Shamsa Kanwal lotus_zone16@yahoo.com
Gerhard Pfister pfister@mathematik.uni-kl.de

Overview: Computing Janet bases and border bases for any ordering using the idea of r-standard bases (defined by V. Gerdt)

References:

[1] A. Kehrein, M. Kreuzer, L. Robbiano: An algebrists view on border bases, in: A. Dickenstein and I. Emiris (eds.), Solving Polynomial Equations: Foundations, Algorithms and Aplications, Springer, Heidelberg 2005, 169-202.

[2] V.P. Gerdt: Involute Algorithms for Computing Groebner Bases, In Computational Commutative and Non-Computational Algebra Geometry, S.Conjocaru, G. Pfister and V. Ufnarovski (Eds.), NATO Science Series,105 Press 2005, 199-255.

Procedures:

`borderBasis(I)`
computes a border basis of the ideal I

`modBorder(I)`
computes a border basis of the ideal I using modular methods

`rJanet(I)`
computes a Janet basis of the ideal I

`modJanet(I)`
computes a Janet basis of the ideal I using modular methods

D.15.34 rwalk_lib

Library: rwalk.lib

Purpose: Groebner Walk Conversion Algorithms

Author: Stephan Oberfranz

Procedures:

`prwalk(ideal,int,int[,intvec,intvec])`
standard basis of ideal via Random Perturbation Walk algorithm

`rwalk(ideal,int[,intvec,intvec])`
standard basis of ideal via Random Walk algorithm

`frandwalk(ideal,int[,intvec,intvec])`
standard basis of ideal via Random Fractal Walk algorithm

See also: Section D.4.7 [grwalk_lib], page 745; Section D.15.36 [swalk_lib], page 866.

D.15.35 schubert_lib

Library: schubert.lib

Purpose: Proceduces for Intersection Theory

Author: Hiep Dang, email: hiep@mathematik.uni-kl.de

Overview: We implement new classes (variety, sheaf, stack, graph) and methods for computing with them. An abstract variety is represented by a nonnegative integer which is its dimension and a graded ring which is its Chow ring. An abstract sheaf is represented by a variety and a polynomial which is its Chern character. In particular, we implement the concrete varieties such as projective spaces, Grassmannians, and projective bundles.

An important task of this library is related to the computation of Gromov-Witten invariants. In particular, we implement new tools for the computation in equivariant intersection theory. These tools are based on the localization of moduli spaces of stable maps and Bott's formula. They are useful for the computation of Gromov-Witten invariants. In order to do this, we have to deal with moduli spaces of stable maps, which were introduced by Kontsevich, and the graphs corresponding to the fixed point components of a torus action on the moduli spaces of stable maps.

As an insightful example, the numbers of rational curves on general complete intersection Calabi-Yau threefolds in projective spaces are computed up to degree 6. The results are all in agreement with predictions made from mirror symmetry computations.

References:

Hiep Dang, Intersection theory with applications to the computation of Gromov-Witten invariants, Ph.D thesis, TU Kaiserslautern, 2013.

Sheldon Katz and Stein A. Stromme, Schubert-A Maple package for intersection theory and enumerative geometry, 1992.

Daniel R. Grayson, Michael E. Stillman, Stein A. Stromme, David Eisenbud and Charley Crissman, Schubert2-A Macaulay2 package for computation in intersection theory, 2010.

Maxim Kontsevich, Enumeration of rational curves via torus actions, 1995.

Procedures:

`makeVariety(int,ideal)`
: create a variety

`printVariety(variety)`
: print procedure for a variety

`productVariety(variety,variety)`
: make the product of two varieties

`ChowRing(variety)`
: create the Chow ring of a variety

`Grassmannian(int,int)`
: create a Grassmannian as a variety

`projectiveSpace(int)`
: create a projective space as a variety

`projectiveBundle(sheaf)`
: create a projective bundle as a variety

`integral(variety,poly)`
: degree of a 0-cycle on a variety

`makeSheaf(variety,poly)`
: create a sheaf

`printSheaf(sheaf)`
: print procedure for sheaves

`rankSheaf(sheaf)`
: return the rank of a sheaf

`totalChernClass(sheaf)`
: compute the total Chern class of a sheaf

`ChernClass(sheaf,int)`
: compute the k-th Chern class of a sheaf

`topChernClass(sheaf)`
: compute the top Chern class of a sheaf

`totalSegreClass(sheaf)`
: compute the total Segre class of a sheaf

`dualSheaf(sheaf)`
: make the dual of a sheaf

`tensorSheaf(sheaf,sheaf)`
: make the tensor of two sheaves

`symmetricPowerSheaf(sheaf,int)`
: make the k-th symmetric power of a sheaf

`quotSheaf(sheaf,sheaf)`
: make the quotient of two sheaves

`addSheaf(sheaf,sheaf)`
: make the direct sum of two sheaves

`makeGraphVE(list,list)`
: create a graph from a list of vertices and a list of edges

`printGraphG(graph)`
: print procedure for graphs

`moduliSpace(variety,int)`
: create a moduli space of stable maps as an algebraic stack

`printStack(stack)`
: print procedure for stacks

`dimStack(stack)`
: compute the dimension of a stack

`fixedPoints(stack)`
: compute the list of graphs corresponding the fixed point components of a torus action on the stack

`contributionBundle(stack,graph)`
: compute the contribution bundle on a stack at a graph

`normalBundle(stack,graph)`
: compute the normal bundle on a stack at a graph

`multipleCover(int)`
: compute the contribution of multiple covers of a smooth rational curve as a Gromov-Witten invariant

`linesHypersurface(int)`
: compute the number of lines on a general hypersurface

`rationalCurve(int,list)`
: compute the Gromov-Witten invariant corresponding the number of rational curves on a general Calabi-Yau threefold

`sumofquotients(stack,list)`
: prepare a command for parallel computation

`homog_part(poly,int)`
: compute a homogeneous component of a polynomial.

`homog_parts(poly,int,int)`
: compute the sum of homogeneous components of a polynomial

`logg(poly,int)`
: compute Chern characters from total Chern classes.

`expp(poly,int)`
: compute total Chern classes from Chern characters

`SchubertClass(list)`
: compute the Schubert classes on a Grassmannian

`dualPartition(list)`
: compute the dual of a partition

D.15.36 swalk_lib

Library: swalk.lib

Purpose: Sagbi Walk Conversion Algorithm

Author: Junaid Alam Khan junaidalamkhan@gmail.com

Overview: A library for computing the Sagbi basis of subalgebra through Sagbi walk algorithm.

Theory: The concept of SAGBI (Subalgebra Analog to Groebner Basis for Ideals) is defined in [L. Robbiano, M. Sweedler: Subalgebra Bases, volume 42, volume 1430 of Lectures Note in Mathematics series, Springer-Verlag (1988),61-87]. The Sagbi Walk algorithm is the subalgebra analogue to the Groebner Walk algorithm which has been proposed in [S. Collart, M. Kalkbrener and D.Mall: Converting bases with the Grobner Walk. J. Symbolic Computation 24 (1997), 465-469].

Procedures:

`swalk(ideal[,intvec])`
: Sagbi basis of subalgebra via Sagbi walk algorithm

`rswalk(ideal,int,int[,intvec])`
: Sagbi basis of subalgebra via Random Sagbi Walk Algorithm

See also: Section D.4.7 [grwalk_lib], page 745; Section D.15.34 [rwalk_lib], page 864.

D.15.37 tropicalNewton_lib

Library: tropicalNewton.lib

Purpose: Computations in Tropical Geometry using Newton Polygon methods

Authors: Tommy Hofman, email: thofmann@mathematik.uni.kl.de Yue Ren, email: reny@cs.bgu.ac.il

Overview: This libraries contains algorithms for computing
- non-trivial points on tropical varieties,
- zero-dimensional tropical varieties,
- one-codimensional links of tropical varieties
based on Newton polygon methods.

References:

Hofmann, Ren: Computing tropical points and tropical links, arXiv:1611.02878 (WARNING: this library follows the max convention instead and triangular sets follow the definition of the Singular book)

Procedures:

`setUniformizingParameter()`
: sets the uniformizingParameter

`val()`
: returns valuation of element in ground field

`newtonPolygonNegSlopes()`
: returns negative of the Newton Polyong slopes

`cccMatrixToPositiveIntvec()`
: helper function to turn a computed valuation vector into a usable weight vector in Singular

`tropicalPointNewton()`
: computes point on tropical variety

`switchRingsAndComputeInitialIdeal()`
: switches rings and computes initial ideal

`tropicalVarietyNewton()`
computes tropical variety of zero-dimensional ideal

`tropicalLinkNewton()`
computes tropical variety that is polyhedral fan and has codimension one lineality space

See also: [tropicalVariety], page 820; Section D.13.4 [tropical_lib], page 817.

8 Release Notes

8.1 News and changes

NEWS in SINGULAR 4-1-1

News for version 4-1-1

New syntax:

`alias`: may be used as a prefix to a variable declaration. Can only be used in procedure headings. (Section 3.5.1 [General command syntax], page 41).

New command:

`fres`: improved version of `sres`: computes a (not necessarily minimal) free resolution of the input ideal/module, using Schreyer's algorithm. (Section 5.1.46 [fres], page 185,Section 5.1.145 [sres], page 264).

Extended commands:

pseudo ordering `L` allows setting of limits for exponents in polynomials (Section B.2.9 [Pseudo ordering L], page 701,Section 5.1.2 [attrib], page 154 for `maxExp`)

`%`,`mod`: also for poly operands (Section 4.16.3 [poly operations], page 120).

`delete`: extended to intvec, ideal, module (Section 5.1.20 [delete], page 168).

syz (Section 5.1.152 [syz], page 275), lift (Section 5.1.78 [lift], page 209), liftstd (Section 5.1.79 [liftstd], page 209), intersect(Section 5.1.63 [intersect], page 200): with a specified GB algorithm

New libraries:

classify2.lib: Classification of isolated singularities of corank `<=`2 and modality `<=` wrt. right equivalence over the complex numbers according to Arnold's list. (Section D.15.5 [classify2_lib], page 832)

goettsche.lib: Goettsche's formula for the Betti numbers of the Hilbert scheme of points on a surface, Macdonald's formula for the symmetric product (Section D.15.14 [goettsche_lib], page 840)

combinat.lib, modules.lib, methods,lib, nets.lib: a more mathematical view of modules (Section D.15.8 [combinat_lib], page 834: combinatorics), (Section D.15.20 [methods_lib], page 846: construct procedures), (Section D.15.21 [modules_lib], page 847: free resolutions), (Section D.15.27 [nets_lib], page 856: pretty printing)

ncHilb.lib: Hilbert series of non-commutative monomial algebras (Section D.15.24 [ncHilb_lib], page 854)

realclassify.lib: Classification of real singularities(Section D.15.31 [realclassify_lib], page 861)

rootisolation.lib: real root isolation using interval arithmetic(Section D.15.32 [rootisolation_lib], page 862)

rstandard.lib: Janet bases and border bases for ideals (Section D.15.33 [rstandard_lib], page 863)

Changed libraries:

chern.lib: new version (Section D.15.3 [chern_lib], page 828)

gitfan.lib: new (incompatible) version (Section D.15.16 [gitfan_lib], page 842)

grobcov.lib: new version (Section D.2.4 [grobcov_lib], page 727)

Changes in the kernel/build system:

port to polymake 3.x.x

port to NTL 10 with threads (needs also C++11: gcc6 or -std=c++11)

p_Invers is only a helper for p_Series: now static

p_Divide is now p_MDivide, pDivive is a new routine

News for version 4-1-0

Syntax changes:

new (additional) form of ring definitions: (for example `ring R=QQ[x,y,z];`) (Section 3.3.2 [General syntax of a ring declaration], page 32)

new (additional) form of multi-indicies: (for example `i(1,2,3,4,5)`) (Section 3.5.3 [Names], page 44)

changed behaviour of `charstr` (Section 5.1.7 [charstr], page 160)

new data type `cring` to describe the coeffient rings, to be used for the new definitions for (polynomial) rings (Section 3.3.2 [General syntax of a ring declaration], page 32)

new command `ring_list` to access the parts used to contruct polynomial rings (Section 5.1.134 [ring_list], page 252,Section 5.1.133 [ringlist], page 251)

extended polynomial ring construction: also from lists produced by `ring_list`

new attribute `ring_cf` for `ring` (Section 5.1.2 [attrib], page 154)

printing of rings changed to match `cring` names (Section 5.1.7 [charstr], page 160)

New libraries:

new library: classifyMapGerms.lib: standard basis of the tangent space at the orbit of an algebraic group action (Section D.15.7 [classifyMapGerms_lib], page 834)

new library: ffmodstd.lib: Groebner bases of ideals in polynomial rings over algebraic function fields(Section D.15.12 [ffmodstd_lib], page 837)

new library: nfmodsyz.lib: syzygy modules of submodules of free modules over algebraic number fields(Section D.15.29 [nfmodsyz_lib], page 859)

new library: curveInv.lib: invariants of curves (Section D.15.9 [curveInv_lib], page 835)

new library: gfan.lib: interface to gfanlib (Section D.13.1 [gfan_lib], page 811)

extended library: interface to polymake merged into Section D.13.2 [polymake_lib], page 814

new library: tropicalNewton.lib: Newton polygon methods in tropical geometry (Section D.15.37 [tropicalNewton_lib], page 867)

new library: schubert.lib: some procedures for intersction theory (Section D.15.35 [schubert_lib], page 864)

Changed libraries:

classify_aeq.lib: new procedures (Section D.15.6 [classify_aeq_lib], page 833)

grobcov.lib: new version (Section D.2.4 [grobcov_lib], page 727)

ncfactor.lib: factorization in some noncommuative algebras (Section 7.7.14 [ncfactor_lib], page 534) with new routine ncfactor (Section 7.7.14.1 [ncfactor], page 535) primdec.lib: new option "subsystem" (Section D.4.21 [primdec_lib], page 756)

Changes in the kernel:

improved mapping of polynomials/ideals/...

port to gcc 6

port to gfanlib 0.6 (requires C++11, i.e. gcc >=4.3)

port to NTL 10

port to polymake 3.0

port to readline 7

Section 5.1.136 [sba], page 254 works for global orderings, also for coefficient types Z and Z/m

Section 5.1.147 [std], page 266 works for all orderings, also for coefficient types Z and Z/m with local/mixed orderings

Section 5.1.35 [factorize], page 178 works for polynomial rings over ZZ

Experimental stuff:

module Section D.14.2 [customstd_lib], page 823: modify `std` ([satstd], page 824)

News for version 4-0-3

New libraries:

new library: brillnoether.lib: Riemann-Roch spaces of divisors on curves (Section D.15.2 [brillnoether_lib], page 827)

new library: chern.lib: Chern classes (Section D.15.3 [chern_lib], page 828)

new library: ffmodstd.lib: Groebner bases of ideals in polynomial rings over algebraic function fields(Section D.15.12 [ffmodstd_lib], page 837)

new library: GND.lib: General Neron Desingularization (Section D.15.17 [GND_lib], page 844)

new library: graal.lib: localization at prime ideals (Section D.15.15 [graal_lib], page 841)

new library: hess.lib: Riemann-Roch space of divisors (Section D.15.19 [hess_lib], page 846)

Changed libraries:

renamed algemodstd_lib to Section D.15.28 [nfmodstd_lib], page 858, extended to `module`

renamed derham_lib to Section D.15.10 [deRham_lib], page 835

grobcov.lib (grobcovK): Groebner Cover for parametric ideals (Section D.2.4 [grobcov_lib], page 727) with new routine ConsLevels ([ConsLevels], page 729), removed AddCons AddConsP.

News for version 4-0-2

New commands:

align (Section 5.1.1 [align], page 154)

branchTo (Section 4.17.3 [procs with different argument types], page 123)

`->` (Section 4.17.2 [proc expression], page 123)

Change in ring handling:

`typeof(` qring `)` returns `"ring"`

New libraries:

algemodstd.lib: Groebner bases of ideals in polynomial rings over algebraic number fields(renamed to Section D.15.28 [nfmodstd_lib], page 858) arr.lib: arrangements of hyperplanes (Section D.14.1 [arr_lib], page 821)

brillnoether.lib: Riemann-Roch spaces of divisors on curve (Section D.15.2 [brillnoether_lib], page 827)

hess.lib: Riemann-Roch space of divisors on function fields and curves (Section D.15.19 [hess_lib], page 846)

gradedModules.lib: graded modules/matrices/resolutions (Section D.15.18 [gradedModules_lib], page 844)

Changed libraries:

revised polymake interface (polymake.so)

revised gfanlib interface (gfanlib.so)

Presolve::findvars ([findvars], page 790, Section 5.1.161 [variables], page 279)

Ring::addvarsTo ([addvarsTo], page 736)

Ring::addNvarsTo ([addNvarsTo], page 737)

Ring::hasAlgExtensionCoefficient ([hasAlgExtensionCoefficient], page 736)

Schreyer::s_res (`s_res`)

grobcov.lib (grobcovK) (Section D.2.4 [grobcov_lib], page 727) with new routines AddCons AddConsP.

normaliz.lib (for normaliz >=2.8) (Section D.4.19 [normaliz_lib], page 753)

renamed groebnerFan to groebnerFanP in polymake.lib (Section D.13.2 [polymake_lib], page 814)

renamed fVector to fVectorP in polymake.lib (Section D.13.2 [polymake_lib], page 814,`polymakeInterface_lib`)

News for version 4-0-1

Version 4-0-1 is a bug fix release.

New feature: attribute `ring_cf` for `ring` (Section 5.1.2 [attrib], page 154)

News for version 4-0-0

Version 4-0-0 is a milestone relase of Singular. The new release series 4 aims for an entirely modularized architecture simplifying connectivity with other systems and paving the way for parallel computations. As a first step in modularization, the new release features an internal structural separation of coefficient rings and polynomial rings. This allows for a flexible integration of new coefficient rings.

Singular 4-0-0's list of new functionality and significant improvements further extends that of the 3-1-6/7 prerelease series.

New functionality

de Rham cohomology of complements of algebraic varieties (Section D.15.10 [deRham_lib], page 835)

Gromov-Witten numbers of elliptic curves (Section D.4.6 [ellipticcovers_lib], page 744)

classification of isolated complete intersection singularities in characteristic 0 (Section D.6.5 [classifyci_lib], page 776)

parametrization of orbits of unipotent actions (Section D.5.3 [orbitparam_lib], page 764)

F5-like Groebner basis algorithm (Section 5.1.136 [sba], page 254) element-wise application of functions to data structures (Section 5.2.1 [apply], page 285)

support for debugging libraries (Section 3.9.1 [ASSUME], page 68)

Improved functionality

Groebner cover for parametric ideals (Section D.2.4 [grobcov_lib], page 727)

normalization of affine rings (Section D.4.18 [normal_lib], page 753)

classification of real singularities (Section D.15.31 [realclassify_lib], page 861)

GIT-fans (Section D.15.16 [gitfan_lib], page 842)

algebraic/transcendental field extensions

Chapter 7 [Non-commutative subsystem], page 312

an abstraction layer for parallel computations (Section D.2.7 [parallel_lib], page 732)

run-time loading of supplementary kernel code (Section A.1.9 [Dynamic modules], page 637)

interpreter language support for name spaces (Section 4.15 [package], page 118)

Availability

Singular is available as source code and for Linux, Mac OS X, Windows, FreeBSD and SunOS-5.

8.2 Singular 3 and Singular 4

The purpose of this section is to describe new features and changes between Singular 3-1-7 and Singular 4.* (formerly known as Spielwiese) both for developers and Singular users. In what follows we will refer to the systems as Singular 3 and Singular 4.

8.2.1 Version schema for Singular

Singular version is of the form `a.b.c.d` which may also be written as `a-b-c-d` where a,b,c and d are numbers:

a is changed with major, incompatible changes

b is changed with incompatible changes (of some commands/libraries)

c is changed with compatible changes (i.e. new commands, extended options, new algorithms, etc.)

d is changed with each release (i.e. with bug fixes, etc.)

Singular does also have "unofficial" build orginating from a code version between "official" version: such builds display "Development version a.b.c" in the header while "official" versions show "version a.b.c". Also the manual describes version a-b-c. To get the complete version number, use `system("version");` or use `SINGULAR_VERSION` in C.

8.2.2 Notes for Singular users

Coefficient rings

To allow for easy integration of new coefficient rings into Singular, the the way coefficient rings are being handled has been redesigned.

In general, the user syntax has not changed, however there are some changes in the behaviour of Singular:

- setting `minpoly` results in changing the current coefficient domain and clears all previously defined varaiables of that ring

- Minor changes in the output of coefficient ring description. Moreover the output of elements of certain rings has been improved (for example, reals).
- Algebraic and transcendental extensions of rationals and finite fields have been reimplemented. In particular, the heuristics for clearing denominators and factoring out content have been changed. In some cases this leads to a different, mathematically equivalent results of Groebner bases and related computations. For example a Groebner basis element may differ by a unit.
- Most noteably, due to the redesign of the coefficient rings, if the user sets the minimal polynomial all variables dependent on the current ring are deleted.

Ring-dependent options

Formally global Singular Section 5.1.108 [option], page 230 now belong to individual polynomial rings. This includes:

- `intStrategy`
- `redTail`
- `redThrough`

Also the following settings now belong to individual (currently active) polynomial rings:

- `short`
- `minpoly`
- `noether`

Hence setting these options only affects the current ring. Be aware of this when switching between different rings, since the options affect the result of various computations (in particular Groebner bases).

Path names

- The tree structure of the binary Singular distribution has been changed. The typical tree now looks as show at `https://github.com/Singular/Sources/wiki/Sw-tree`
- Accordingly Singular search paths (where Singular searches for libraries, dynamic modules, etc.) have been changed. You can display them by calling Singular by `Singular -v`.
- currently, multi-arch installations of Singular 4 aere not possible.

Library versioning

Due to switching from Subversion to GIT revision control system for the Singular source code, library version variables (displayed when loading a library) have changed.

New orderings for modules

The now can assign weights to module components, when defining a monomial ordering. For example

```
ring R = 0, (x,y,z), (am(1,2,3,   10,20,30,40), dp, C);
deg(x*gen(1));
↦ 11
```

will assign weights 1,2,3 to x,y,z respectively, and weights 10,20,30,40,0,0,... to components of any free module defined over R. This ordering will first sort by this weighted degree, then by dp on the ring monomials and then will give priority to the large component index.

Future benefits of Singular 4

The redesign of Singular will allow us to provide new features in the future, for example:

- Interpreter type for coefficient rings.
- User defined coefficient rings.
- Improved syntax for defining polynomial rings.

8.2.3 Notes for developers

There has been an entensive process of refactoring, redesign and modularization of Singular to facilitate easier maintenance and future development:

- Build System : automake, libfac has been integrated into Factory
- Removed MP (Multi protocol) in favor of SSI links.
- Separation/modularization into libraries and packages
- For easy integration of new coeffcient rings, we defined a generic interface for coefficient rings and a supporting framework for making them accessible to the user.

  In particular we have separated everything related to coeffcient rings into a separate library `libcoeffs`. Dependency tree between restructured packages is show at `http://www.singular.uni-kl.de/dox/singular.png`

  In order to use `libSingular` as a C++ library, see Section 8.3 [libSingular], page 875.

8.2.4 Building Singular

The user can build and install Singular with the following standard UNIX-like procedure:

- Download and extract the latest official source package (.tar.gz).
- Run the configure script, for instance, `./configure`.
- Build Singular by running `make`.
- Install Singular by running `make install`.

In contrast to Singular 3, there are now many more configuration options.

All possible options for configure can be seen by running the configure script with option `--help`. On a multicore compute consider running make with the option `-j [cores]`.

8.2.5 Side-by-side installation

Due to choosing paths according to FS standards it is no longer possible to have a side-by-side installation of different Singular versions or versions for different architectures.

8.3 libSingular

`libSingular` is the C++-library version of SINGULAR.
`Singular/libsingular.h` is the main include file, `-lSingular` the link parameter,
`lib/pkgconfig/Singular.pc` provides all parameters in the pkconfig format.

It contains all parts of SINGULAR with two exceptions:

1. memory allocation functions for GMP (see `mmInit` in `Singular/tesths.cc`)
2. signal handlers (see `init_signals` in `Singular/cntrlc.cc`).
   At least a handler for `SIGCHLD` must be installed for the routines from Section D.2.7 [parallel_lib], page 732, Section D.4.13 [modstd_lib], page 748, Section D.4.12 [modnormal_lib],

page 748, Section D.2.12 [tasks_lib], page 737.
If the child was started by `libSingular` the handler has to call `sig_chld_hdl` from `Singular/links/ssiLink.cc` or implement something similiar (call `slClose(l)` for ssi links).

8.4 Download instructions

Singular is available as source and binary program for most common hard- and software platforms. Instructions to download and install Singular can be found at

`http://www.singular.uni-kl.de/index.php/singular-download.html`.

Release versions of Singular are also available from our FTP site

`ftp://www.mathematik.uni-kl.de/pub/Math/Singular/src/4-0-3/`.

8.5 Unix installation instructions

Install binaries: `http://www.singular.uni-kl.de/index.php/singular-download/install-linuxunix.html`

or build it yourself:

Install the necessary packages:

- libtool
- gnu make
- gcc, g++
- libreadline
- gmp
- ntl
- libcdd

Install flint 2.5: `./configure --with-gmp=/usr --prefix=$HOME/tmp --disable-shared`
`make && make install`

Install Singular `./configure --with-flint=$HOME/tmp --enable-gfanlib --prefix=$HOME/Singular4`
`make && make install`
(`$prefix/bin/Singular` is the main execuatble)

(optional) install 4ti2

(optional) install surf/surfer

(optional) install normaliz 2.8 (or newer)

See also `https://github.com/Singular/Sources/wiki/Step-by-Step-Installation-Instructions-for-Singular` which includes instructions adapted for debian and fedora based systems.

8.6 Windows installation instructions

Singular relies on Cygwin as its environment under Windows. There is a 32bit and a 64bit version of Cygwin.

`http://www.singular.uni-kl.de/index.php/singular-download/install-windows.html`

8.7 Macintosh installation instructions

Installation of the provided binaries `http://www.singular.uni-kl.de/index.php/singular-download/install-mac-os.html`

9 Index

‘

_

{

|

}

~

"

+

>

^

\

<

A

C

D

E

F

G

H

I

M

N

O

P

Q

R

S

T

U

V

Table of Contents