1 Preface

SINGULAR version 3-0-2
University of Kaiserslautern
Department of Mathematics and Centre for Computer Algebra
Authors: G.-M. Greuel, G. Pfister, H. Schoenemann

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

Please register yourself as a SINGULAR user by using the registration form on the SINGULAR homepage `http://www.singular.uni-kl.de`. If for some reason you cannot access the registration form, you can also register 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).

If you use Singular or parts thereof in a project and/or publish results that were partly obtained using SINGULAR, we ask you to cite SINGULAR and inform us thereof – see `http://www.singular.uni-kl.de/how_to_cite.html`, for information on how to cite Singular.

Availability

The latest information about SINGULAR is always available from `http://www.singular.uni-kl.de`.

Acknowledgements

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

Currently, the SINGULAR team has the following members: Michael Brickenstein, Wolfram Decker, Anne Frühbis-Krüger, Kai Krüger, Christoph Lossen, Viktor Levandovskyy, Wilfred Pohl and Oliver Wienand.

Former members of the SINGULAR team are: Olaf Bachmann, Wolfgang Neumann, Jens Schmidt, Mathias Schulze, Thomas Siebert, Rüdiger Stobbe, Eric Westenberger and Tim Wichmann.

Further contributions to SINGULAR were made by: Thomas Bayer, Isabelle Bermejo, Markus Becker, Kai Dehmann, Alexander Dreyer, Stephan Endraß, Jose Ignacio Farran, Vladimir Gerdt, Philippe Gimenez, Christian Gorzel, Hubert Grassmann, Fernando Hernando, Agnes Heydtmann, Dietmar Hillebrand, Tobias Hirsch, Thomas Keilen, Simon King, Anen Lakhal, Martin Lamm, Santiago Laplagne, Gregoire Lecerf, Francisco Javier Lobillo, Bernd Martin, Michael Meßollen, Andrea Mindnich, Jorge Martin Morales, Oleksandr Motsak, Thomas Nüßler, Carlos Rabelo, Alfredo Sanchez-Navarro, Henrik Strohmayer, Christian Stussak, Imade Sulandra, Christine Theis, Enrique Tobis, Alberto Vigneron-Tenorio, Moritz Wenk, Denis Yanovich, 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.

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 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 factorizations, 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 compute.

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 (much faster standard and Groebner bases computations based on Hilbert series and on improved implementations of the algorithms, libraries for primary decomposition, ring normalization, etc.)

For the highlights of the new SINGULAR version 3-0-2 see ⟨undefined⟩ [News and changes], page ⟨undefined⟩.

2.2 How to use this tutorial

In Chapter 3 [Getting started], page 5, some simple examples explain how to use SINGULAR in a step-by-step manner.

Chapter 4 [Examples], page 14 should come next for real learning-by-doing or to quickly solve some given mathematical problems without dwelling too deeply into SINGULAR.

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 be set using the command `setring`.

- the arrow ↦ 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 control key together with the key `P` to get the previous input line)

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 are available without a ring.

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 done by the symbol `=` .

```
int k = 2;
```

Test for equality resp. inequality is done using `==` resp. `!=` (or `<>`), where `0` represents the boolean value FALSE, 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 is always available with 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;` . Explanation on single topics, e.g., on `intmat`, which defines a matrix of integers, are obtained by

```
help intmat;
```

This shows the text from node `intmat`, 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 a ring being active. 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.

3.2 Rings and standard bases

To calculate with objects 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. So the example above 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 means that the degree reverse lexicographical ordering should 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 extended precision of 50 digits, variable names `a` and `b`.

`ring r7=(complex,50,i),(a,b),lp;`
: complex floating point numbers with extended precision of 50 digits and imaginary unit `i`, 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 r8;
r8;
↦ //   characteristic : 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 the chapter "Rings and orderings" of the SINGULAR manual.

The basering is now `r8`. Since we want to calculate in the ring `r`, which we defined first, we have 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 `^`, saying `x^3`, or in short-hand notation without operator, saying `x3`. Note that the short-hand notation is forbidden if a name of the ring variable(s) consists of more than one character(see ⟨undefined⟩ [Miscellaneous oddities], page ⟨undefined⟩ 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` determines in general the number of "single entries" in an object. In particular, for polynomials, `size` determines 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.

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 4.4.2 [Critical points], page 40, 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 have 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, then we 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
```

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 collect several commands to a new one. Procedures are defined with the keyword `proc` followed by a name and an optional parameter list with specified types. Finally, a procedure may return values using the command `return`.

Define the following procedure called `Milnor`:

```
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 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 `help "all.lib";` lists all libraries together with a one-line explanation.

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 with 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.

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. Define the localization of the polynomial ring at the origin.

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

This ordering determines the standard basis which will be calculated. Fetch the polynomial defined in the ring `r` into this new ring, thus avoiding retyping the 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.

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`, 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 form.

```
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 computations 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.

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 give 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 represented 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.

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 the description 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 determined 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.

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 $(-xyz - y^2z - 4xz^2 + 16z^3, -y^2, 48z, 48z, x+y-z)$.

4 Examples

4.1 Programming

4.1.1 Basic programming

We show in the example below the following:

- define the ring R with char 32003, variables x,y,z, monomial ordering dp (implementing F_32003[x,y,z])
- list the information about R 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 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 ⟨undefined⟩ [primdec_lib], page ⟨undefined⟩)
- 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 ⟨undefined⟩ [imap], page ⟨undefined⟩) - 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 Chapter 3 [Getting started], page 5.

```
ring R = 32003,(x,y,z),dp;
R;
↦ //   characteristic : 32003
↦ //   number of vars : 3
↦ //        block   1 : ordering dp
↦ //                  : names    x y z
↦ //        block   2 : ordering C
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
```

4.1.2 Writing procedures and libraries

The user may add his 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 ⟨undefined⟩ [Procedures], page ⟨undefined⟩, and ⟨undefined⟩ [Format of a library], page ⟨undefined⟩.

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 ⟨undefined⟩ [Debugging tools], page ⟨undefined⟩).

We give short examples of procedures to demonstrate the following:

- Write procedures which return an integer (ring independent), see also Section 4.4.1 [Milnor and Tjurina number], page 39. (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]:
```

```
↦      1.791287847477920003294023596864
```

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 4.1.6 [Dynamic modules], page 21.

4.1.3 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 ⟨undefined⟩ [Monomial orderings], page ⟨undefined⟩ 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 ⟨undefined⟩ [division], page ⟨undefined⟩ for division with remainder in the localization and ⟨undefined⟩ [invunit], page ⟨undefined⟩ for a procedure returning a truncated power series expansion of the inverse of a unit.

4.1.4 Parameters

Let us now deform a given 0-dimensional ideal j by introducing a parameter t and compute over the ground field Q(t). We compute the dimension at the generic point, i.e., $dim_{Q(t)} Q(t)[x,y]/j$.

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;
↦ //   characteristic : 0
↦ //   1 parameter    : t
↦ //   minpoly        : 0
↦ //   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
```

4.1.5 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
```

4.1.6 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. Detailed information on the latter topic can be found in a separate PostScript file at `http://www.singular.uni-kl.de/DynMod.ps`.

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);
↦ x2-y2,xy+y2,y2,
↦ 1,    0,    0,
↦ 0,    1,    0,
↦ 0,    0,    1

// 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);
↦ // Kstd                 [0]  package (C,kstd.so)
↦ // Standard             [0]  package (S,standard.lib)
↦ // Top                  [0]  package (N)

// 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);
↦ 0,  0,  y2,xy,x2,
↦ -y, y,  0, 0, 1,
↦ x-y,-x, 0, 1, 0,
↦ 0,  x+y,1, -1,1

// 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);
↦ 0,  0,   y2,xy,x2,
↦ 0,  y,   0, 0, 1,
↦ -y, -x+y,0, 1, 0,
↦ x+y,0,   1, -1,1
```

4.2 Computing Groebner and Standard Bases

4.2.1 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 ⟨undefined⟩ [option], page ⟨undefined⟩).

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 4.2.2 [Groebner basis conversion], page 23, Section 4.2.3 [slim Groebner bases], page 25. For information on the heuristics behind `groebner`, see the library file `standard.lib` (see also Section 3.3 [Procedures and libraries], page 9).

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 ⟨undefined⟩ [Cyclic roots], page ⟨undefined⟩). We set `option(prot)` to make SINGULAR display some information on the performed computations (see ⟨undefined⟩ [option], page ⟨undefined⟩ 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);
↦ [31:1]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(27)\
   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(56)\
   s(59)s(62)s(63)s(65)s(66)s(68)s(70)s(73)s(75)s(78)---ss(81)-----------s(7\
   3)--------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(101\
   )s(108)s(110)----------(100)--------9-s(94)s(97)s(99)s(84)s(74)s(77)s(80)\
   ---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(73)\
   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----[1023:2]36-s37(6)s38s39s40---42-s43(5)s44s45--48-s49s50s51---54-s55\
   (4)--67-86-
↦ product criterion:664 chain criterion:2844
timer-t;                                // used time (in 1/100 secs)
↦ 79
size(sI);
↦ 17
t=timer;
sI=groebner(I);
↦ std in (32003),(a,b,c,d,e,f,@t@),(dp(7),C)
```

```
↦ [15:1]1(5)s2(4)s3(3)s4ss5(4)s(6)s(7)s(9)6-s(11)s(12)s(14)s(16)s(19)s(21)s\
  (24)-7-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 (32003),(a,b,c,d,e,f,@t@),(lp(7),C)
↦ [15:1]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(27)\
  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(56)\
  s(59)s(62)s(63)s(65)s(66)s(68)s(70)s(73)s(75)s(78)---ss(81)-----------shh\
  hhhhhhh8(64)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(101\
  )s(108)shhhhhhhhhhhhhhhhhhh9(91)s(94)s(97)s(99)s(125)s(128)s(131)s(134)s(\
  147)s(152)s(157)s(161)s(162)s(164)s(166)s(169)---s-s(170)-----------s(163\
  )---------shhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhh10(124)s(127)s(129)s(132)s(138\
  )s(141)s(144)s(147)shhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhh\
  hh11(99)s(102)s(104)s(107)s(110)shhhhhhhhhhhhhhhhhhhhhhhhhhhh12(85)s(88)s\
  (95)s(98)s(102)--(100)-----------shhhhhhhhhhhhhh13(78)s(82)s(85)s(87)s(90\
  )s(93)----shhhhhhhhhhhhhhhhhhhhhhhhhhh14(65)s(70)s(73)s(77)s(80)s(83)----\
  sshhhhhhhhhhhhhhhhhhhhhhhhhhhhhhh15(57)s(61)s(64)s(68)s(74)----shhhhhhhhh\
  hhhhhhhhhhhhhhhhhhhhhh16(42)s(45)s(49)shhhhhhhhhhhhhhhhhhhhhh17(34)s(37)s\
  hhhhhhhhhhhhhhh18(27)s(30)s(32)[255:2]-shhhhhhhh19(26)s(29)s(32)shhhhhhhh\
  hhhhh20(22)s(25)s(28)shhhhhhhhhhhhh21(20)s(26)shhhhhhhhhhhh22(18)shhhhhhh\
  hh23(12)shhhhh24(11)s(14)-shhhh25(13)s(18)-s(21)shhhhhhh26(18)shhhhhhhhhh\
  hh27(9)shhhhh28(8)shhhh29(7)shhhh30(8)-shhh31shhhhh32(7)shhhh33shhhh34(6)\
  shhhhhhhh36(2)s37(6)shhhh38shhhh39shhhhhhhh42(2)s43(5)shhhh44shhhhhhhh48s\
  49shhhh50shhhhhhhh54shhhhh
↦ product criterion:720 chain criterion:11620
↦ hilbert series criterion:448
↦ dehomogenization
↦ imap to original ring
↦ simplification
timer-t;                                // used time (in 1/100 secs)
↦ 28
size(sI);
↦ 17
option(noprot);
```

4.2.2 Groebner basis conversion

The performance of Buchberger's algorithm is sensitive to the choice of 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 ⟨undefined⟩ [stdfglm], page ⟨undefined⟩) and variants of the Groebner walk conversion algorithm (which works for arbitrary ideals, See ⟨undefined⟩ [frwalk], page ⟨undefined⟩, ⟨undefined⟩ [grwalk_lib], page ⟨undefined⟩). An implementation of the Hilbert-driven Buchberger algorithm is accessible via the `stdhilb` command (see also ⟨undefined⟩ [std], page ⟨undefined⟩).

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);
↦ [63:2]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(1\
  7)--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);
↦ [31:1]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(1\
  7)--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(23)\
  s(26)-s(27)------s(23)----------13-s(15)-----------14-s(6)--15-s(5)--16--\
  -
↦ product criterion:88 chain criterion:650
↦ [31:1]1(4)s2(3)s3(2)s4s(3)s5(5)s(8)s(10)s-ss6(12)s(15)s(17)s(20)s(21)s(22\
  )s(24)s(25)--shhh7(23)s(25)s(27)s(29)s(31)-s(32)s(38)s(40)s(41)s(44)--shh\
  hhhhhhhh8(33)s(36)s(40)s(43)s(46)-s(47)s(50)--------shhhhhhhhhhhh9(32)s\
  (34)s(37)s(39)s(42)s(45)------s(42)s(45)shhhhhhhhhh10(38)s(40)s(43)s(47)\
  s(49)s(52)s(55)--s(56)------s(53)s(56)s(58)--shhhhhhhhhhhhhh11(45)s(47)s(\
```

```
  50)s(53)s(56)s(59)s(61)---s-s(63)s(66)----s(65)-------shhhhhhhhhhhhhhhhh1\
  2(44)s(47)s(50)s(51)s(53)s(56)s(58)s(61)s(63)s(66)s(69)s(72)--s(73)----sh\
  hhhhhhhhhhhhhhhhhhhhhhhhh13(45)s(48)s(51)s(54)s(56)s(58)s(61)s(64)shhhhhh\
  hhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhh14(29)s(31)s(34)s(37)s(40)s(43)shhhhhhhhh\
  hhhhhhhhhhhhhhh15(22)s(25)s(28)s(31)s(34)shhhhhhhhhhhhhhhhhh16(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)shhhhhhhh\
  h21(11)s(14)s(17)shhhhhhhhh22(11)s(14)s(16)shhhhhhhhh23(10)s(13)shhhhhhhh\
  h24(7)s(10)shhhhhh25(7)s(10)shhhhhh26(7)s(10)shhhhhh27(7)s(10)shhhhhh28(7\
  )s(10)shhhhhh29(7)s(10)shhhhhh30(7)s(9)shhhhhh31(6)shhhhhh[1023:2]32(3)sh\
  hh33shhh34shhh35shhh36shhh37shhh38shhh39shhh40shhh41shhh42shhh43shhh44shh\
  h45shhh46shhh47shhh48shhh49shhh50shhh51shhh52shhh53shhh54shhhhhh
↦ product criterion:491 chain criterion:11799
↦ hilbert series criterion:382
↦ map[63:1]{2833:2944}----------...
 timer-t;
↦ 1
 size(j2);   // size (no. of polys) in computed GB
↦ 158
 // usual Groebner basis computation for lex ordering
 t=timer;
 ideal j0 =std(i);
↦ [63:1]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(1\
  1)s(13)s8(12)s(13)s(15)s.s(14).s.9.s(16)s(17)s(19)........10.s(20).s(21)s\
  s..11.s(23)s(25).ss(27)...s(28)s(26)...12.s(25)sss(23)sss.......s(22)...1\
  3.s(23)ssssssss(21)s(22)sssss(21)ss..14.ss(22)s.s.sssss(21)s(22)sss.s...1\
  5.ssss(21)s(22)ssssssssss(21)s(22)sss16.ssssssss(21)s(22)sssssssssss17ss(\
  21)s(22)ssssssssss(21)sss(22)ss(21)ss18(22)s(21)s(22)s.s..............19.\
  sssss(21)ss(22)ssssssssss(21)s(22)s20.ssssssssss(21)s........21.s(22)ssss\
  sssssssssss(21)s(22)ssss22ssssssssssss(21)s(22)sssssss23sssssssssss(21)s(\
  22)ssssssss24ssssssssssss(21)s(22)sssssss25ssssssssss(21)s(22)sssssssss26\
  ssssssssss(21)s(20)ssssssss27.sssssssss..........s28.ssssss.............2\
  9.sssssssssssssssssss30sssssssssssssssssss31.sssssssssssssssssss32.ssssss\
  ssssssssssssss33ssssssssssssssssssss34ssssssssssssssssssss35sssssssssssss\
  sssssss36ssssssssssssssssssss37ssssssssssssssssssss38ssssssssssssssssssss\
  39ssssssssssssssssssss40ssssssssssssssssssss41ssss---------------42-s(4)-\
  -43-s44s45s46s47s48s49s50s51s52s53s54s55s56s
↦ product criterion:1395 chain criterion:904
 option(noprot);
 timer-t;
↦ 1
```

4.2.3 slim Groebner bases

The command `slimgb` 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 and with respect to degree orderings.

In the example below (Groebner basis with respect to degree reverse lexicographic ordering over function field) it is more than 600 times 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);
↦ //used time: 0.69 sec
```

For detailed information and limitations see ⟨undefined⟩ [slimgb], page ⟨undefined⟩.

4.3 Commutative Algebra

4.3.1 Saturation

For any two ideals i, j in the basering R let

$$\text{sat}(i,j) = \{x \in R \mid \exists\, n \text{ s.t. } x \cdot (j^n) \subseteq i\} = \bigcup_{n=1}^{\infty} i : j^n$$

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 4.4.2 [Critical points], page 40). 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 iff $sat(j + (f), m)$ is the whole ring, that is, generated by 1. (Note that `maxideal(n)` denotes the n-th power of the maximal ideal).

```
  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
```

4.3.2 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 ⟨undefined⟩ [Extra weight vector], page ⟨undefined⟩), 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 4.2.2 [Groebner basis conversion], page 23):

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 the 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 two 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]=x
↦ j2[2]=z2-y3
  // 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
↦ i[4]=x3z
  //
  // 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 = H-J;         // this is tricky: difference between two
                         // ideals is not defined, but between two
                         // matrices. By automatic 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+12ac2e-2abde
  // 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
```

4.3.3 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` 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.

`lres` computes a free resolution of an ideal or module with LaScala's method. The input needs to be homogeneous.

`mres` computes a minimal free resolution of an ideal or module with the syzygy method.

`sres` computes a free resolution of an ideal or module with Schreyer's method. The input has to be a standard basis.

`nres` computes a free resolution of an ideal or module with the standard basis method.

`minres` minimizes a free resolution of an ideal or module.

`syz` 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's 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
↦ resolution not minimized yet
↦
  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
```

4.3.4 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 apllied when computing data from free resolutions (see ⟨undefined⟩ [Computation of Ext], page ⟨undefined⟩).

4.3.5 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
```

4.3.6 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
```

4.3.7 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 `vectors` 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
```

4.3.8 Algebraic dependence

Let g, f_1, ..., $f_r \in K[x_1,\ldots,x_n]$. We want to check whether

1. f_1, ..., 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, ..., 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 blockordering 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.
```

4.4 Singularity Theory

4.4.1 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. Look at the manual to get information about the possible monomial orderings in SINGULAR, or type `help Monomial orderings;` to get a menu of possible orderings. For further help type, e.g., `help local 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` with char 32003, 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 ⟨undefined⟩ [sing_lib], page ⟨undefined⟩ for the library commands for the computation of the Milnor and Tjurina number.

```
  option(prot);
  ring r1 = 32003,(x,y,z),ds;
  r1;
↦ //   characteristic : 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);
↦ [1023:2]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;    // overwrite j
  j=std(j);
↦ [1023:2]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);
↦ [1023:2]7(2)s8s10s11s12s13(3)ss(4)s14(5)s(6)s15(7).....s(8)16.s...s(9)..1\
   7............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
↦ [1023:2]7(16)s8(15)s10s11ss(16)-12.s-s13s(17)s(18)s(19)-s(18).-14-s(17)-s\
   (16)ss(17)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..\
   .......sH(19)16(5).....18......19.-----
↦ product criterion:15 chain criterion:174
↦ 195
  option(noprot);
```

4.4.2 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 4.4.1 [Milnor and Tjurina number], page 39 and its elements.

The following will be implemented below:

- reset the protocol option and activate the timer
- define the ring `r2` with char 32003, 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 4.4.1 [Milnor and Tjurina number], page 39) 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
```

4.4.3 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 his 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 give three examples, the first being a hypersurface, the second a complete intersection, the third no complete intersection and compute in each of the cases the miniversal deformation:

```
  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: (32003),(A,B,C,D,E,F,x,y,z),(ds(6),ds(3),C);
↦ // 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]=xy+Ez+F
↦ Fs[1,2]=y2+z2+x3+Ay+Bx2+Cx+D
  // 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.
↦ // ** J is no standard basis
↦
↦
↦ // '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 }
  //---------------------------------------------------
```

4.4.4 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 ⟨undefined⟩ [hnoether_lib], page ⟨undefined⟩) 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 definbed 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 squarefree 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
```

4.4.5 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 a 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.
```

4.5 Invariant Theory

4.6 Noncommutative Algebra

4.6.1 Left and two-sided Groebner bases

For a set of polynomials (resp. vectors) `S` in a noncommutative 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 ⟨undefined⟩ [std (plural)], page ⟨undefined⟩). 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 ⟨undefined⟩ [twostd], page ⟨undefined⟩).

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 ⟨undefined⟩ [vdim (plural)], page ⟨undefined⟩).
Further on, we use the command `reduce` (see ⟨undefined⟩ [reduce (plural)], page ⟨undefined⟩) 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 ⟨undefined⟩ [option], page ⟨undefined⟩ and ⟨undefined⟩ [Groebner bases in G-algebras], page ⟨undefined⟩ 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 ⟨undefined⟩ [option], page ⟨undefined⟩ for an interpretation of the displayed symbols).

```
// ----- 1.  setting up the algebra
  ring A = 0,(e,f,h),dp;
  matrix D[3][3];
  D[1,2]=-h; D[1,3]=2*e; D[2,3]=-2*f;
  ncalgebra(1,D);
// ----- 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)s
↦ (7)s
↦ 3(6)s
↦ (8)(7)(6)4(5)(4)(3)(2)s
↦ (3)(2)s
↦ 3(3)(2)45
↦ (S:5)-----
↦ product criterion:6 chain criterion:16
  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
```

4.6.2 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;
↦ //   characteristic : 0
↦ //   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 A = 0,(e,f,h),dp;
// matrix D[3][3];
// D[1,2]=-h; D[1,3]=2*e; D[2,3]=-2*f;
// ncalgebra(1,D);
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);
↦ -2e2h+6e2,
↦ -12ef2+24fh+12f,
↦ -4e2fh-2e2f,
↦ -4e3fh+16e3f-6e2h2+42e2h-72e2
// --- let us define the opposite algebra of A
def Aop = opposite(A);
setring Aop; Aop;         // see how Aop looks like
↦ //   characteristic : 0
↦ //   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
```

4.7 Applications

4.7.1 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. Or, to find a system of polynomials defining an ideal which has the same radical as I (see Section 4.2 [Computing Groebner and Standard Bases], page 21, resp. ⟨undefined⟩ [radical], page ⟨undefined⟩).

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, a first step in a SINGULAR computation could be to reduce the number of generators by applying the `interred` command (see ⟨undefined⟩ [interred], page ⟨undefined⟩). 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 ⟨undefined⟩ [triang_lib], page ⟨undefined⟩) for the ideal I, or by applying the factorizing Buchberger algorithm (see ⟨undefined⟩ [facstd], page ⟨undefined⟩), or by computing a primary decomposition of I (see ⟨undefined⟩ [primdec_lib], page ⟨undefined⟩).

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 ⟨undefined⟩ [sat], page ⟨undefined⟩).

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 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*2.449489742783178098197284074706)
↦    [5]:
↦       (i*1.732050807568877293527446341506)
↦ [2]:
↦    [1]:
↦       1
↦    [2]:
↦       1
↦    [3]:
↦       1
↦    [4]:
↦       (-i*2.449489742783178098197284074706)
↦    [5]:
↦       (i*1.732050807568877293527446341506)
↦ [3]:
↦    [1]:
↦       1
↦    [2]:
↦       1
↦    [3]:
↦       1
↦    [4]:
↦       (i*2.449489742783178098197284074706)
↦    [5]:
↦       (-i*1.732050807568877293527446341506)
↦ [4]:
↦    [1]:
↦       1
↦    [2]:
↦       1
↦    [3]:
↦       1
↦    [4]:
↦       (-i*2.449489742783178098197284074706)
↦    [5]:
↦       (-i*1.732050807568877293527446341506)
↦ 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*2.449489742783178098197284074706)
↦ [5]:
↦    (-i*1.732050807568877293527446341506)
```

4.7.2 AG codes

The library `brnoeth.lib` provides an implementation of the Brill-Noether algorithm for solving the Riemann-Roch problem and applications in 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 evaluating 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 ith 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 **
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 to the
// support of D:
intvec F=6;                   // F = 6*point #1
// in F, the ith entry refers to the ith 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
//
//  Coding and Decoding:
matrix word[1][11];           // words of length 11 are coded
```

```
word = 1,1,1,1,1,1,1,1,1,1,1,1;
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
```

4.8 Further smallexamples

The example section of the SINGULAR manual contains further examples, e.g.:

- Long coefficients
  how they arise in innocent smallexamples
- T1 and T2
  compute first order deformations and obstructions
- Finite fields
  compute in fields with $q = p^n$ elements
- Ext
  compute Ext groups, derived from the Hom functor
- Polar curves
  compute local and global polar curves
- Depth
  various ways to compute the depth of a module
- Cyclic roots
  create and compute with this standard benchmark smallexample

- Invariants of finite group
  compute invariant rings for finite group
- Normalization
  compute the normalization of a ring
- Classification of hypersurface singularities
  determine type and normal form of a hypersurface singularity after Arnold
- Parallelization with MPtcp links
  use MP for distributed and parallel computation

In this list the names of the items are the names of the examples in the online help system. So by the command `help T1 and T2` the example about the computation of first order deformations and obstructions is displayed.

Table of Contents