B. SINGULAR — A Short Introduction

De computer is niet de steen, maar de slijpsteen der wijzen.
(The computer is not the philosophers' stone but the philosophers' whetstone.)
Hugo Battus, Rekenen op taal (1983).

In this section we shall give a short introduction to the computer algebra system SINGULAR. In all the chapters of this book there are already many examples for the interactive use of SINGULAR and the possibility to write own programmes in the SINGULAR programming language. For more details we refer to the SINGULAR Manual, which is offered as online help for SINGULAR, and can be found as a postscript file in the distribution. See also the SINGULAR examples in the distribution. We start with instructions on how to obtain SINGULAR, explain the first steps on how to work with SINGULAR and, finally, give an overview about the data types and functions of SINGULAR.

B.1 Downloading Instructions

SINGULAR is available, free of charge, as a binary programme for most common hardware and software platforms. Release versions of SINGULAR can be downloaded through ftp from our FTP site

ftp://www.mathematik.uni-kl.de/pub/Math/Singular/,

or, using your favourite WWW browser, from

http://www.mathematik.uni-kl.de/ftp/pub/Math/Singular/.

To download SINGULAR *for an* RPM-*based Unix platform*:[1]

Make sure that you have approximately 20 MByte of free disk space under /opt and follow these steps:

[1] To install surf, you may either download surf-<version>-ix86-Linux.tar.gz from the enclosed CD and extract the files to /opt. Or you may download the source code from http://sourceforge.net/projects/surf and compile it, following the instructions in the INSTALL file.

(1) There are the following Singular RPM packages:
   - `Singular-<version>-1.i386.rpm` — Singular (dynamically linked) executables `Singular` and `ESingular` (requires shared libraries)
   - `Singular-static-<version>-1.i386.rpm` — Singular (statically linked) executables `Singular` and `ESingular` (does not require shared libraries)
   - `Singular-share-<version>-1.noarch.rpm` — Singular libraries, documentation, and examples
   - `Singular-redhat-<version>-1.noarch.rpm` — Gnome and KDE support for RedHat Linux
   - `Singular-suse-<version>-1.noarch.rpm` — Gnome and KDE support for SuSE Linux

(2) Choose the RPM packages you need and download them.

(3) To install or deinstall RPM packages, you need to log in as a superuser (type `su` in a command shell and enter the root password).

(4) As a superuser, you can install the Singular RPM packages using a graphical RPM frontend like e.g. `gnorpm` under Gnome or `kpackage` under KDE.[2] Alternatively, you can use the RPM command line tool `rpm`. To install an RPM package `Singular-...-<version>-1.i386.rpm` using `rpm`, type `rpm -ivh Singular-...-<version>-1.i386.rpm` in a command shell.

To download Singular *for a non* RPM-*based Unix platform*:[3]

Make sure that you have approximately 20 MByte of free disk space and follow these steps (you don't need superuser privileges):

(1) You need to download two (archive) files:
   - `Singular-<version>-share.tar.gz` — architecture independent data such as documentation and libraries
   - `Singular-<version>-<uname>.tar.gz` — architecture dependent executables, such as the Singular programme

   Here, `<uname>` can, for example, be one of the following:
   - `ix86-Linux` — PC's running under Linux with `libc` version 6 (e.g. SuSE Linux distribution version ≥ 6 or RedHat Linux distribution version ≥ 5.2)
   - `ix86-Linux-libc5` — PC's running under Linux with `libc` version 5 (run `ldd /bin/ls` to find out your `libc` version)
   - `HPUX-9` — HP workstations running under HPUX version 9
   - `HPUX-10` — HP workstations running under HPUX version 10

[2] Under Gnome or KDE, simply double-click on the icon of a Singular RPM package to start the standard RPM application.

[3] To install `surf`, you may either download `surf-<version>-ix86-Linux.tar.gz` from the CD and extract the files to the directory where you installed Singular. Or download the source code from `http://sourceforge.net/projects/surf` and compile it, following the instructions in the `INSTALL` file.

- SunOS-5 — Sun workstations running Solaris version 5

(2) Simply change to the directory into which you wish to install Singular (usually wherever you install third–party software):

```
cd /usr/local/
```

Singular specific subdirectories will be created in such a way that multiple versions and multiple architecture dependent files of Singular can peaceably coexist under the same /usr/local/ tree.

(3) Unpack the archives:

```
gzip -dc Singular-<version>-<uname>.tar.gz | tar -pxf -
gzip -dc Singular-<version>-share.tar.gz | tar -pxf -
```

This creates the following (sub)directories, which contain

Singular/<version>/<uname>	Singular and ESingular programmes
Singular/<version>/LIB	Singular libraries (*.lib files)
Singular/<version>/emacs	Singular Emacs user interface
Singular/<version>/info	info files of Singular manual
Singular/<version>/html	html files of Singular manual
Singular/<version>/doc	miscellaneous documentation files
Singular/<version>/examples	Singular examples (*.sing files)

To download Singular *for Windows 95/98/ME/NT/2000/XP*:

(1) Download one of the following self–extracting archives:

Singular-<version>-Compact.exe
Minimal archive to download. Installs Singular , minimal set of needed tools and the Singular Manual in WinHelp format.

Singular-<version>-Typical.exe
Typical archive to download. Installs Singular with necessary tools, the XEmacs editor, ESingular and the Singular Manual in WinHelp and Html formats.

Double-click (or execute) the self–extracting archives, and *carefully* follow the instructions given there.

To download Singular *for the Macintosh*:

There are two different possibilities:

(1) For MacOS X take <uname>=ppcMac-darwin and follow the instructions for a Unix platform.

(2) For older MacOS systems Singular is an MPW–tool and the last distribution for such systems.
Make sure that you have MPW installed on your system. Download the file Singular-<version>-ppc-MPW.sea-hqx. Expand the downloaded file using a standard expander to obtain the corresponding self–extracting Singular-<version>-ppc-MPW.sea file. Extract this file. This will create

a new folder named "Singular:" which contains the 'Singular' programme in the subfolder "`<version>:ppc-MPW:`". Do not change the structure of the subfolders. Make the folder which contains the executable the working directory of the MPW shell. To start SINGULAR, you need to type in 'Singular' in the MPW worksheet.

B.2 Getting Started

SINGULAR can either be run in text terminal or within Emacs.

To start SINGULAR in its text terminal user interface, enter `Singular` (or `Singular-<version>`) at the system prompt. The SINGULAR banner appears which, among others, reports the version (and the compilation date, search path, location of programme and libraries, etc.):

```
                     SINGULAR                             /
 A Computer Algebra System for Polynomial Computations   /   version <version>
                                                        0<
     by: G.-M. Greuel, G. Pfister, H. Schoenemann        \   April 2002
 FB Mathematik der Universitaet, D-67653 Kaiserslautern   \
```

To start SINGULAR in its Emacs user interface enter `ESingular` at the system prompt.

Generally, we recommend using SINGULAR in its Emacs interface, since this offers many more features and is more convenient to use than the ASCII–terminal interface (in particular, if running under Windows).

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

There are a few important notes which one should keep in mind:

- Every command has to be terminated by a `;` (semicolon) followed by a RETURN.
- The online help is accessible by means of `help;` or `?;`.

SINGULAR is a special purpose system for polynomial computations. Hence, most of the powerful computations in SINGULAR require the prior definition of a ring. The 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.
Once SINGULAR is started, it awaits an input after the prompt `>`. Every statement has to be terminated by `;` . If you type `37+5;` followed by a `Return`, then `42` will appear on the screen. We always start the SINGULAR output with `//->` to have it clearly separated from the input.

```
37+5;
//-> 42
```

All objects have a type, for example, integer variables are defined by the word `int`. An assignment is done by the symbol `=` .

```
int k = 2;
```

Testing for equality, respectively inequality, is done using `==`, respectively `!=`, (or `<>`), where `0` represents the boolean value FALSE, any other integer 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*_;      // two times the value of k displayed above
//-> 4
```

Starting with `//` denotes a comment and the rest of the line 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.

To edit SINGULAR input we have the following possibilities:

`TAB:` automatic name completion.
`CTRL-B:` moves the cursor to the left.
`CTRL-F:` moves the cursor to the right.
`CTRL-D:` deletes the symbol in the cursor.
`CTRL-P:` gives the preceding line in the history.
`RETURN:` sends the current line to the SINGULAR–Parser.

For more commands see the SINGULAR Manual.

The SINGULAR Manual is available online by typing the command `help;`. Explanations on single topics, for example, on `intmat`, which defines a matrix of integers, are obtained by

```
help intmat;
```

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;
```

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 `{`, respectively `}`, denote the beginning, respectively 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
```

`j++` is equivalent to `j=j+1`. To count the even and the odd entries of the second row of this matrix, we use the `if` and `else` and a `while` loop.

```
j=0;
int even,odd;
while(j<3)
{
  j++;
  if((m[2,j] mod 2)==0){even++;}
  else                 {odd++;}
}
even;
//-> 2
odd;
//-> 1
```

Variables of type `string` can also be defined and used without a ring being active. Strings are delimited by `"` (quotation–marks). 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 were 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:";    //+ concatenates strings
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.

To read data from a file, or to write it to a file, we can use the commands `read` and `write`. For example, the following SINGULAR session creates a file `hallo.txt` and writes 2 and `example` to this file.

```
int a = 2;
write("hallo.txt",a,"example");
```

This data can be read from the file again (as strings):

```
read("hallo.txt");
//-> 2
//-> example
```

If we want to execute commands from a file, we have to write

```
execute(read("hallo.txt"));
```

or, in short,

```
<"hallo.txt";
```

If, for instance, in `hallo.txt` we have the command

```
int j=1; j;
```

we obtain

```
//-> 1
```

To calculate with objects as polynomials, ideals, matrices, modules, and polynomial vectors, a ring has to be defined first:

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

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 (that is, the rational numbers) and ring variables called `x`, `y`, and `z`. The `lp` at the end means that the lexicographical ordering is used.

The default ring in SINGULAR is $\mathbb{Z}/32003[x,y,z]$ with degree reverse lexicographical ordering:

```
ring s;
s;
//-> //   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 basering is now `s`. If 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 command `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 shorthand notation without operator, saying `x3`. Note that the shorthand notation is forbidden if the name of the ring variable consists of more than one character. Note also 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
```

B.3 Procedures and Libraries

SINGULAR offers a comfortable programming language, with a syntax close to C. So it is possible to define procedures which combine several commands to form 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`.

Assume we want to compute for $f = x^3y^2 - x^2y^3 + x^3 + y^3 + z^2 \in \mathbb{Q}[x,y,z]$ the vector space dimension $\dim_{\mathbb{Q}}(\mathbb{Q}[x,y,z]/\langle \partial f/\partial x, \partial f/\partial y, \partial f/\partial z\rangle)$, the so–called *global Milnor number* of f (see also Page 487). Then we type

```
ring  R=0,(x,y,z),dp;
poly  f=x3y2-x2y3+x3+y3+z2;
ideal J=diff(f,x),diff(f,y),diff(f,z);
```

or

```
ideal J=jacob(f);
```

Then we have to compute a standard basis of J:

```
ideal K=std(J);
```

We obtain the Milnor number as

```
vdim(K);
//-> 12.
```

As a procedure, we can write

```
proc Milnor (poly h)
{
  ideal J=jacob(h);
  ideal K=std(J);
  int   d=vdim(K);
  return(d);
}
```

or, in a more compact form:

```
proc Milnor (poly h)
{
 return(vdim(std(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
```

If we want to compute the local Milnor number of f at 0 (cf. Definition A.9.3), then we can use the same procedure, but we have to define a local monomial ordering on R, for example, `ring R=0,(x,y,z),ds;`, and then define f, J, K, as above.

The distribution of SINGULAR contains several libraries, which extend the functionality of SINGULAR. Each of these libraries is a collection of useful procedures based on the kernel commands. 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
```

Let us compute the local Milnor number of f. We use the command `imap` to map f from R to the new basering with a local ordering:

```
ring S=0,(x,y,z),ds;
poly f=imap(R,f);
milnor(f);
//-> 4
```

(Since the global Milnor number is the sum of all local Milnor numbers at all critical points of f, this shows that f must have further critical points outside 0.)

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.

B.4 Data Types

In this section, we give an overview of all data types in SINGULAR. For more details see the Manual.

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

ideal: Ideals are represented as lists of polynomials, which generate the ideal: `ideal I=x2-1,xy;`. Like polynomials, they can only be defined or accessed with respect to a basering.

ideal operations are:

- `+` addition (concatenation of the generators and simplification),
- `*` multiplication (with ideal, poly, vector, module; simplification in case of multiplication with ideal),
- `^` exponentiation (by a nonnegative integer).

ideal related functions are:

`char_series`, `coeffs`, `contract`, `diff`, `degree`, `dim`, `eliminate`, `facstd`, `factorize`, `fglm`, `finduni`, `groebner`, `highcorner`, `homog`, `hilb`, `indepSet`, `interred`, `intersect`, `jacob`, `jet`, `kbase`, `koszul`, `lead`, `lift`, `liftstd`, `lres`, `maxideal`, `minbase`, `minor`, `modulo`, `mres`, `mstd`, `mult`, `ncols`, `preimage`, `qhweight`, `quotient`, `reduce`, `res`, `simplify`, `size`, `sortvec`, `sres`, `std`, `stdfglm`, `stdhilb`, `subst`, `syz`, `vdim`, `weight`

int: Variables of type int represent the machine integers and are, therefore, limited in their range (for example, the range is between -2147483647 and 2147483647 on 32–bit machines): `int i=1;` .

int operations are:

`++`	changes its operand to its successor, is itself not an int expression
`--`	changes its operand to its predecessor, is itself not an int expression
`+`	addition
`-`	negation or subtraction
`*`	multiplication
`/`	integer division (omitting the remainder), rounding toward 0
`div`	integer division (omitting the remainder `>=0`)
`%`	integer modulo (the remainder of the division `/`)
`mod`	integer modulo (the remainder of the division `div`), always nonnegative
`^`, `**`	exponentiation (exponent must be nonnegative)
`<`, `>`, `<=`, `>=`, `==`,`<>`	comparison

An assignment `j=i++;` or `j=i--;` is not allowed, in particular it does not change the value of `j`.

int related functions are:

`char`, `deg`, `det`, `dim`, `extgcd`, `find`, `gcd`, `koszul`, `memory`, `mult`, `ncols`, `npars`, `nrows`, `nvars`, `ord`, `par`, `pardeg`, `prime`, `random`, `regularity`, `rvar`, `size`, `trace`, `var`, `vdim`.

boolean expressions are:

A boolean expression is really an int expression used in a logical context: an int expression (`<>` 0 evaluates to *TRUE* (represented by 1), 0 represents *FALSE*).

boolean operations are:

`and` (logical), may also be written as `&&`,
`or` (logical), may also be written as `||`,
`not` (not logical), may also be written as `!`

intmat: Integer matrices are matrices with integer entries. Integer matrices do not belong to a ring, they may be defined without a basering being defined: `intmat im[2][3]=1,2,3,4,5,6;`.

intmat operations are:

- `+` addition with intmat or int; the int is converted into a diagonal intmat
- `-` negation or subtraction with intmat or int; the int is converted into a diagonal intmat
- `*` multiplication with intmat, intvec, or int; the int is converted into a diagonal intmat
- `div,/` division of entries in the integers (omitting the remainder)
- `%, mod` entries modulo int (remainder of the division)
- `<>`, `==` comparison

intmat related functions are:

`betti`, `det`, `ncols`, `nrows`, `random`, `size`, `transpose`, `trace`.

intvec: Variables of type intvec are lists of integers: `intvec iv=1,2,3,4;`.

intvec operations are:

- `+` addition with intvec or int (component–wise)
- `-` change of sign or subtraction with intvec or int (component–wise)
- `*` multiplication with int (component–wise)
- `/`, `div` division by int (component–wise)
- `%, mod` modulo (component–wise)
- `<>`, `==`, `<=`, `>=`, `>`, `<`
  comparison (done lexicographically)

intvec related functions are:

`hilb`, `indepSet`, `leadexp`, `nrows`, `qhweight`, `size`, `sortvec`, `transpose`, `weight`.

link: Links are the communication channels of SINGULAR, that is, something SINGULAR can write to and/or read from. Currently, SINGULAR supports four different link types:

- ASCII links,
- MPfile links,
- MPtcp links,
- DBM links.

link related functions are:

`close`, `dump`, `getdump`, `open`, `read`, `status`, `write`, `kill`.

Via ASCII links, data, which can be converted to a string, can be written into files for storage or communication with other programmes. The data is written in plain text format. The output format of polynomials is done w.r.t. the value of the global variable `short`. Reading from an ASCII link returns a string — conversion into other data is up to the user. This can be done, for example, using the command `execute`.

MP (Multi Protocol) links give the possibility to store and communicate data in the binary MP format: read and write access is very fast compared to ASCII links.

MPtcp links give the possibility to exchange data in the binary MP format between two processes, which may run on the same or on different computers.
DBM links provide access to data stored in a data base.

list: Lists are arrays whose elements can be of any type (including ring and qring). If one element belongs to a ring the whole list belongs to that ring: `list L=5,"Hallo";`.
Note that a list stores the objects themselves and not the names. Hence, if `L` is a list, `L[1]` for example has no name. A name, say `R`, can be created for `L[1]` by `def R=L[1];`.

list operations are:

`+`	concatenation
`delete`	deletes one element from list, returns new list
`insert`	inserts or appends a new element to list, returns a new list

list related functions are:

`bareiss`, `betti`, `delete`, `facstd`, `factorize`, `insert`, `lres`, `minres`, `mres`, `names`, `res`, `size`, `sres`.

map: Maps are ring maps from a ring into the basering: `map g=R,x,y,z;`.

map operations are:

composition of maps. If, for example, `f` and `g` are maps, then `f(g)` is a map expression giving the composition of `f` and `g`.

matrix: Objects of type matrix are matrices with polynomial entries. Like polynomials they can only be defined or accessed with respect to a basering: `matrix m[2][2]=x,1,y,xy;`.

matrix related functions are:

`bareiss`, `coef`, `coeffs`, `det`, `diff`, `jacob`, `koszul`, `lift`, `liftstd`, `minor`, `ncols`, `nrows`, `print`, `size`, `subst`, `trace`, `transpose`, `wedge`.

module: Modules are submodules of a free module over the basering with basis `gen(1),gen(2), ...` . They are represented by lists of vectors, which generate the submodule. Like vectors they can only be defined or accessed with respect to a basering. If M is a submodule of R^n, R the basering, generated by vectors $v_1, \ldots, v_k$, then $v_1, \ldots, v_k$ may be considered as the generators of relations of R^n/M between the canonical generators `gen(1), ..., gen(n)`. Hence, any finitely generated R–module can be represented in Singular by its module of relations:
`module mo=[x,yz,y],[xy,1,3];`.

module operations are:

`+`	addition (concatenation of the generators and simplification)
`*`	multiplication with ideal or poly.

module related functions are:

`coeffs`, `degree`, `diff`, `dim`, `eliminate`, `freemodule`, `groebner`, `hilb`, `homog`, `interred`, `intersect`, `jet`, `kbase`, `lead`, `lift`, `liftstd`, `lres`, `minbase`, `modulo`, `mres`, `mult`, `ncols`, `nrows`, `print`, `prune`, `qhweight`, `quotient`, `reduce`, `res`, `simplify`, `size`, `sortvec`, `sres`, `std`, `subst`, `syz`, `vdim`, `weight`.

number: Numbers are elements from the coefficient field (or ground field). They can only be defined or accessed with respect to a basering, which determines the coefficient field: `number n=4/6;`.

number operations are:

`+`	addition
`-`	change of sign or subtraction
`*`	multiplication
`/`	division
`^`, `**`	power, exponentiation (by an integer)
`<=`, `>=`, `==`, `<>`	comparison.

number related functions are:

`cleardenom`, `impart`, `numerator`, `denominator`, `leadcoef`, `par`, `pardeg`, `parstr`, `repart`.

poly: Polynomials are the basic data for all main algorithms in SINGULAR. They consist of finitely many terms, which are combined by the usual polynomial operations. Polynomials can only be defined or accessed with respect to a basering, which determines the coefficient type, the names of the indeterminants and the monomial ordering.

```
ring r=32003,(x,y,z),dp;
poly f=x3+y5+z2;
```

poly operations are:

`+`	addition
`-`	change of sign or subtraction
`*`	multiplication
`/`	division by a monomial, non divisible terms yield 0
`^`, `**`	power by an integer
`<`, `<=`, `>`, `>=`, `==`, `<>`	comparison (w.r.t. the monomial ordering)

poly related functions are:

`cleardenom`, `coef`, `coeffs`, `deg`, `det`, `diff`, `extgcd`, `factorize`, `finduni`, `gcd`, `homog`, `jacob`, `lead`, `leadcoef`, `leadexp`, `leadmonom`, `jet`, `ord`, `qhweight`, `reduce`, `rvar`, `simplify`, `size`, `subst`, `trace`, `var`, `varstr`.

proc: Procedures are sequences of SINGULAR commands in a special format. They are used to extend the set of SINGULAR commands with user defined commands. Once a procedure is defined it can be used like any other SINGULAR command. Procedures may be defined by either typing them on the command line or by loading them from a file. A file containing procedure definitions, which comply with certain syntax rules is called a library. Such a file is loaded using the command `LIB`.

qring: SINGULAR offers the opportunity to calculate in quotient rings, that is, rings modulo an ideal. The ideal has to be given as a standard basis: `qring Q=I;`.

resolution: The resolution type is intended as an intermediate representation, which internally retains additional information obtained during computation of resolutions. It furthermore enables the use of partial results to compute, for example, Betti numbers or minimal resolutions. Like ideals and modules, a resolution can only be defined w.r.t. a basering.

resolution related functions are:

`betti`, `lres`, `minres`, `mres`, `res`, `sres`.

ring: Rings are used to describe properties of polynomials, ideals, etc. Almost all computations in SINGULAR require a basering.

ring related functions are:

`charstr`, `keepring`, `npars`, `nvars`, `ordstr`, `parstr`, `qring`, `setring`, `varstr`.

ring operations are:

`+`	construct a new ring $k[X,Y]$ from $k[X]$ and $k[Y]$.

string: Variables of type `string` are used for output (almost every type can be "converted" to `string`) and for creating new commands at runtime. They are also return values of certain interpreter related functions. String constants consist of a sequence of any characters (including newline!) between a starting `"` and a closing `"` . There is also a string constant `newline`, which is the newline character. The `+` sign "adds" strings, `""` is the empty string. Strings may be used to comment the output of a computation or to give it a nice format. Strings may also be used for intermediate conversion of one type into another.
A comma between two strings makes an expression list out of them (such a list is printed with a separating blank in between), while a `+` concatenates strings.

string operations are:

`+`	concatenation
`<=`, `>=`, `==`, `<>`	comparison (lexicographical with respect to the ASCII encoding)

string related functions are:

`charstr`, `execute`, `find`, `names`, `nameof`, `option`, `ordstr`, `parstr`, `read`, `size`, `sprintf`, `typeof`, `varstr`.

vector: Vectors are elements of a free module over the basering with basis `gen(1)`, `gen(2)`, ... Each vector belongs to a free module of rank equal to the greatest index of a generator with non–zero coefficient. Since generators with zero coefficients need not be written, any vector may be considered also as an element of a free module of higher rank. (For example, if `f` and `g` are polynomials then `f*gen(1)+g*gen(3)+gen(4)` may also be written as `[f,0,g,1]` or as `[f,0,g,1,0]`.) Like polynomials they can only be defined or accessed with respect to the basering. Note that the elements of a vector have to be surrounded by square brackets (`[ , ]`).

vector operations are:

`+`	addition
`-`	change of sign or subtraction
`/`	division by a monomial, not divisible terms yield 0
`<`, `<=`, `>`, `>=`, `==`, `<>`	comparison of leading terms w.r.t. monomial ordering

vector related functions are:
`cleardenom`, `coeffs`, `deg`, `diff`, `gen`, `homog`, `jet`, `lead`, `leadcoef`, `leadexp`, `leadmonom`, `nrows`, `ord`, `reduce`, `simplify`, `size`, `subst`.

B.5 Functions

This section gives a short reference of all functions, commands and special variables of the SINGULAR kernel (that is, all built–in commands).
The general syntax of a function is

```
[target =] function_name (<arguments>);
```

If no target is specified, the result is printed. In some cases (for example, `export`, `keepring`, `kill`, `setring`, `type`) the brackets enclosing the arguments are optional. For the commands `help`, `continue`, `break`, `quit` and `exit` brackets are not allowed.

attrib: `attrib (` name `)` displays the attribute list of the object called name. `attrib (` name `,` string `)` returns the value of the attribute string of the variable name. If the attribute is not defined for this variable, `attrib` returns the empty string. `attrib (` name `,` string `,` expression `)` sets the attribute string of the variable name to the value expression.
An attribute may be described by any string. Some of these are used by the kernel of SINGULAR and referred to as reserved attributes.
Reserved attributes are: `isSB`, `isHomog`, `isCI`, `isCM`, `rank`, `withSB`, `withHilb`, `withRes`, `withDim`, `withMult`.

bareiss: `bareiss(M)` applies the sparse Gauß-Bareiss algorithm to a module (or with type conversion to a matrix) M using column operations with an “optimal” pivot strategy. It returns a module corresponding to a lower triangular matrix and a vector of integers giving the permutation of the rows made during the algorithm. Since no division by pivot elements takes place, it works over $K[x_1, \ldots, x_n]$, for example, for computing the determinant.

betti: `betti(L)` computes the graded Betti numbers of R^n/M (R the basering, M a homogeneous submodule of R^n) from the given resolution L of R^n/M.

char: `char(R)` returns the characteristic of the coefficient field of the ring R.

char_series: `char_series(I)` computes a matrix. The rows of the matrix represent the irreducible characteristic series (cf. [132]) of the ideal I with respect to the current ordering of variables.

charstr: `charstr(R)` returns the description of the coefficient field of the ring R as a string.

cleardenom: `cleardenom(f)` multiplies the polynomial f, respectively vector f, by a suitable constant to cancel all denominators of the coefficients and then divides it by its content.

close: `close(L)` closes the link L.

coef: `coef(f,m)` determines the monomials in f divisible by one of the variables appearing in m (which is a product of ring variables) and the coefficients of these monomials as polynomials in the remaining variables.

coeffs: `coeffs(J,z)` develops each polynomial of J as a univariate polynomial in the given ring variable z, and returns the coefficients as a $k \times d$ matrix M, where $d-1$ is the maximal z–degree of all occurring polynomials; k is the number of generators if J is an ideal ($k = 1$, if J is a polynomial). If J is a vector or a module this procedure is repeated for each component and the resulting matrices are appended. An (optional) third argument T is used to return the matrix T of coefficients such that `matrix`$(\mathrm{J}) = TM$.
`coeffs(M,K,p)` returns a matrix A of coefficients with $KA = M$ such that the entries of A do not contain any variable from p. Here K is a set of monomials, respectively vectors, with monomial entries, in the variables appearing in p, p is a product of variables (if this argument is not given, then the product of all ring variables is taken as default argument).
M is supposed to consist of elements of (respectively have entries in) a finitely generated module over a ring in the variables not appearing in p. K should contain the generators of M over this smaller ring.
If K does not contain all M, then $KA = M'$ where M' is the part of M corresponding to the monomials of K.

contract: `contract(I,J)` contracts each of the n elements of the second ideal J by each of the m elements of the first ideal I, producing an $m \times n$–matrix. Contraction is defined on monomials by:

$$\mathrm{contract}(x^A, x^B) := \begin{cases} x^{B-A}, & \text{if } B \geq A \text{ component–wise} \\ 0, & \text{otherwise} \end{cases}$$

where A and B are the multi–exponents of the ring variables represented by x. `contract` is extended bilinearly to all polynomials.

defined: `defined(name)` returns a value <> 0 *(TRUE)* if there is a user–defined object with this name, and 0 *(FALSE)* otherwise.

deg: `deg(f)` returns the maximal (weighted) degree of the terms of a polynomial or a vector f; the weights are the weights used for the first block of the ring ordering. deg(0) is -1.

degree: `degree(I)` computes the (weighted) degree of the projective variety, respectively sheaf over the projective variety, defined by the ideal, respectively module, generated by the leading monomials of the input. This is equal to the (weighted) degree of the projective variety, respectively sheaf over the projective variety, defined by the ideal, respectively module, if the input is a standard basis with respect to a (weighted) degree ordering.

delete: `delete(L,n)` deletes the n–th element from the list L.

diff: `diff(f,x)` computes the partial derivative of the polynomial f by the ring variable x.

dim: `dim(I)` computes the dimension of the ideal, respectively module, generated by the leading monomials of the given generators of the ideal I, respectively module. This is also the dimension of the ideal if it is represented by a standard basis.

division: computes a division with remainder. `division(I,J)` returns a list `T,R,U` where `T` is a matrix, `R` is an ideal, respectively a module, and `U` is a diagonal matrix of units such that `matrix(I)*U=matrix(J)*T+matrix(R)` is a standard representation for the (weak) normal form `R` of `I` with respect to a standard basis of `J`.

dump: `dump(L)` dumps (that is, writes in one "message" or "block") the state of the SINGULAR session (that is, all defined variables and their values) to the link L (which must be either an ASCII or MP link) such that a `getdump` can retrieve it later on.

eliminate: `eliminate(I,m)` eliminates variables occurring as factors of the second argument m from an ideal I, respectively module, by intersecting it with the subring not containing these variables. `eliminate` does not need a special ordering nor a standard basis as input.
Since elimination is expensive, for homogeneous input it might be useful first to compute the Hilbert function of the ideal (first argument) with a fast ordering (for example, `dp`). Then make use of it to speed up the computation: a Hilbert–driven elimination uses the intvec provided as the third argument.

eval: evaluates (quoted) expressions. Within a quoted expression, the quote can be "undone" by an `eval` (that is, each `eval` "undoes" the effect of exactly one quote). Used only when receiving a quoted expression from an MPfile link, with `quote` and `write` to prevent local evaluations when writing to an MPtcp link.

ERROR: immediately interrupts the current computation, returns to the top level, and displays the argument `string_expression` as error message. This should be used as an emergency, respectively failure, exit within procedures.

example: `example topic;` computes an example for `topic`. Examples are available for all SINGULAR kernel and library functions.

execute: executes a string containing a sequence of SINGULAR commands.

exit: exits (quits) SINGULAR, works also from inside a procedure or from an interrupt.

extgcd: computes extended gcd: the first element is the greatest common divisor of the two arguments, the second and third are factors such that if `list L=extgcd(a,b);` then L[1]=a*L[2]+b*L[3].
Polynomials must be univariate to apply `extgcd`.

facstd: `facstd(I)` returns a list of ideals computed by the factorizing Gröbner basis algorithm. The intersection of these ideals has the same zero–set as the ideal I, that is, the radical of the intersection coincides with the radical of the input ideal. In many (but not all!) cases this is already a decomposition of the radical of the ideal. (Note, however, that, in general, no inclusion between the input and output ideals holds.) A second, optional, argument can be a list of polynomials, which define non–zero constraints. Hence, the intersection of the output ideal has a zero–set, which is the (closure of the) complement of the zero–set of the second argument in the zero–set of the first argument.

factorize: `factorize(f)` computes the irreducible factors (as an ideal) of the polynomial f together with or without the multiplicities (as an intvec) depending on the optional second argument:
- 0: returns factors and multiplicities, first factor is a constant. (`factorize(f)` is a short notation for `factorize(f,0)`).
- 1: returns non–constant factors (no multiplicities).
- 2: returns non–constant factors and multiplicities.

fetch: `fetch(R,I)` maps the object I defined over the ring R to the basering. `fetch` is the canonical map between rings and qrings: the i–th variable of the source ring R is mapped to the i–th variable of the basering. The coefficient fields must be compatible.
Compared to `imap`, `fetch` uses the position of the ring variables, not their names.

fglm: `fglm(R,I)` computes for the ideal I in the ring R a reduced Gröbner basis in the basering, by applying the so–called FGLM (Faugère, Gianni, Lazard, Mora) algorithm. The main application is to compute a lexicographical Gröbner basis from a reduced Gröbner basis with respect to a degree ordering. This can be much faster than computing a lexicographical Gröbner basis directly. The ideal must be zero–dimensional and given as a reduced Gröbner basis in the ring R.

fglmquot: `fglmquot(I,p)` computes a reduced Gröbner basis of the ideal quotient $I:\langle p\rangle$ of a zero–dimensional ideal I and an ideal generated by a polynomial p, by using FGLM–techniques. The ideal must be zero–dimensional and given as a reduced Gröbner basis in the given ring. The polynomial must be reduced with respect to the ideal.

find: returns the first position of a substring in a string or 0 (if not found), starts the search at the position given in the (optional) third argument.

finduni: `finduni(I)` returns an ideal, which is contained in the given ideal I such that the i–th generator is a univariate polynomial in the i–th ring variable. The polynomials have minimal degree with respect to this property. The ideal must be zero–dimensional and given as a reduced Gröbner basis in the current ring.

freemodule: `freemodule(n)` creates the free module of rank n generated by `gen(1),...,gen(n)`.

gcd: computes the greatest common divisor of two integers or two polynomials.

gen: `gen(i)` is the i–th free generator of a free module.

getdump: `getdump(L)` reads the content of the entire file, respectively link, L and restores all variables from it. For ASCII links, `getdump` is equivalent to an `execute(read(` link `))` command. For MP links, `getdump` should only be used on data, which were previously `dump`ed.

groebner: `groebner(I)` computes the standard basis of the argument `I` (ideal or module), by a heuristically chosen method: if the ordering of the current ring is a local ordering, or if it is a non–block ordering and the current ring has no parameters, then `std(I)` is returned. Otherwise, `I` is mapped into a ring with no parameters and ordering dp, where its Hilbert series is computed. This is followed by a Hilbert series based standard basis computation in the original ring.
If a second argument `wait` is given, then the computation proceeds at most `wait` seconds. That is, if no result could be computed in `wait` seconds, then the computation is interrupted, 0 is returned, a warning message is displayed, and the global variable `groebner_error` is defined.

help: `help topic;` displays online help information for `topic` using the currently set help browser. If no `topic` is given, the title page of the manual is displayed.

- `?` may be used instead of `help`.
- `topic` can be an index entry of the SINGULAR manual or the name of a (loaded) procedure, which has a help section.
- `topic` may contain wildcard characters (that is, `*` characters).
- If a (possibly "wildcarded") `topic` cannot be found (or uniquely matched) a warning is displayed and no help information is provided.
- If `topic` is the name of a (loaded) procedure whose help section has changed w.r.t. the help available in the manual then, instead of displaying the respective help section of the manual in the help browser, the "newer" help section of the procedure is simply printed to the terminal.
- The browser in which the help information is displayed can be set either with the command–line option
  `--browser=<browser>`
  or, if SINGULAR is already running, with the command

`system("--browser", "<browser>")`.
Use the command
`system("browsers");`
for a list of all available browsers.

highcorner: `highcorner(I)` returns the smallest monomial not contained in the ideal, respectively module, generated by the initial terms of the given generators of I. If the generators are a standard basis, this is also the smallest monomial not contained in the ideal, respectively module I. If the ideal, respectively module, is not zero–dimensional, 0 is returned. Hence, `highcorner` is always 1 or 0 for global monomial orderings.

hilb: computes the (weighted) Hilbert series of the ideal, respectively module, defined by the leading terms of the generators of the given ideal, respectively module. If `hilb(I)` is called with one argument (the ideal or module I), then the first and second Hilbert series, together with some additional information, are displayed. If `hilb(I,n)` is called with two arguments, then the n–th Hilbert series is returned as an intvec, where $n = 1, 2$ is the second argument. More precisely, if `hilb(I,n)` $= v_0, \ldots, , v_d, 0$, then the n–th Hilbert series ($Q(t)$, respectively $G(t)$) is $\sum_{i=0}^{d} v_i t^i$, cf. page 278.
If a weight vector w is a given as third argument, then the Hilbert series is computed with respect to these weights w (by default all weights are set to 1).
The last entry of the returned intvec is not part of the actual Hilbert series, but is used in the Hilbert driven standard basis computation.
If the input is homogeneous with respect to the weights and a standard basis, the result is the (weighted) Hilbert series of the original ideal, respectively module.

homog: `homog(I)` tests for homogeneity: returns 1 for homogeneous input, 0 otherwise. `homog(I,t)` homogenizes polynomials, vectors, generators of ideals or modules I by multiplying each monomial with a suitable power of the given ring variable t (which must have weight 1).

hres: computes a free resolution of a homogeneous ideal using the Hilbert driven algorithm. More precisely, let R be the basering and I be the given ideal. Then `hres(I,k)` computes a minimal free resolution of R/I

$$\ldots \to F_2 \xrightarrow{A_2} F_1 \xrightarrow{A_1} R \to R/I \to 0\,.$$

If k is not zero then the computation stops after k steps and returns a list of modules $M_i =$`module`$(A_i), i = 1 \ldots k$.
`list L=hres(I,0);` returns a list L of n modules (where n is the number of variables of the basering) such that $L[i] = M_i$ in the above notation.

imap: is the map between rings and qrings with compatible ground fields, which is the identity on variables and parameters of the same name and 0 otherwise. `imap(R,i)` maps I defined over the ring R to the basering. Compared with `fetch`, `imap` uses the names of variables and parameters. Unlike `map` and `fetch`, `imap` can map parameters to variables.

impart: returns the imaginary part of a number in a complex ground field, returns 0 otherwise.

indepSet: `indepSet(I)` computes a maximal set U of independent variables of the ideal I given by a standard basis. If v is the result then $v[i]$ is 1 if and only if the i–th variable of the ring, $x(i)$, is an independent variable. Hence, the set U consisting of all variables $x(i)$ with $v[i] = 1$ is a maximal independent set.

insert: `insert(L,I)` inserts a new element I into a list L at the first place or (if called with three arguments) after the given position.

interred: `interred(I)` interreduces a set of polynomials or vectors I .

intersect: `intersect(I,J)` computes the intersection of the ideals, respectively modules, I and J.

jacob: `jacob(f)` computes the Jacobian ideal, respectively Jacobian matrix, generated by all partial derivatives of the input f.

jet: `jet(f,k)` deletes from the first argument, f, all terms of degree larger than the second argument, k. If a third argument, w, of type intvec is given, the degree is replaced by the weighted degree defined by w. `jet` is independent of the given monomial ordering.

kbase: `kbase(I)` computes a vector space basis (consisting of monomials) of the quotient ring by the ideal, respectively of a free module by the module, I, in case it is finite dimensional and if the input is a standard basis with respect to the ring ordering.
With two arguments: computes the part of a vector space basis of the respective quotient with degree of the monomials equal to the second argument. Here, the quotient does not need to be finite dimensional.

kill: deletes objects.

killattrib: deletes the attribute given as the second argument

koszul: `koszul(d,n)` computes a matrix of the Koszul relations of degree d of the first n ring variables. `koszul(d,id)` computes a matrix of the Koszul relations of degree d of the generators of the ideal `id`. `koszul(d,n,id)` computes a matrix of the Koszul relations of degree d of the first n generators of the ideal `id`.

laguerre: `laguerre(p,n,m)` computes all complex roots of the univariate polynomial p using Laguerre's algorithm. The second argument, n, defines the precision of the fractional part if the ground field is the field of rational numbers, otherwise it will be ignored. The third argument (can be 0, 1 or 2) gives the number of extra runs for Laguerre's algorithm (with corrupted roots), leading to better results.

lead: `lead(I)` returns the leading term(s) of a polynomial, a vector, respectively of the generators of an ideal or module I with respect to the monomial ordering.

leadcoef: `leadcoef(f)` returns the leading coefficient of a polynomial or a vector f with respect to the monomial ordering.

leadexp: `leadexp(f)` returns the exponent vector of the leading monomial of a polynomial or a vector f. In the case of a vector the last component is the index in the vector.

leadmonom: `leadmonom(f)` returns the leading monomial of a polynomial or a vector f as a polynomial or vector, whose coefficient is one.

LIB: reads a library of procedures from a file. If the given filename does not start with `.` or `/` and cannot be located in the current directory, each directory contained in the `SearchPath` for libraries is searched for a file of this name.

lift: `lift(m,sm)` computes the transformation matrix, which expresses the generators of a submodule in terms of the generators of a module. More precisely, if m denotes the module (or ideal), if sm denotes the submodule (or subideal), and if T denotes the transformation matrix returned by lift, then `matrix(sm)*U=matrix(m)*T`, where U is a diagonal matrix of units. U is always the unity matrix if the basering is a polynomial ring (not a power series ring). U is stored in the optional third argument.

liftstd: `liftstd(m,T)` returns a standard basis of an ideal or module and the transformation matrix from the given ideal, respectively module, to the standard basis. That is, if m is the ideal or module, sm the standard basis returned by `liftstd`, and T the transformation matrix, then `matrix(sm)=matrix(m)*T`.

listvar: lists all (user–)defined names in the current namespace:

• `listvar()`:	all currently visible names except procedures,
• `listvar(type)`:	all currently visible names of the given type,
• `listvar(ring_name)`:	all names, which belong to the given ring,
• `listvar(name)`:	the object with the given name,
• `listvar(all)`:	all names except level of variables in procedures is shown in square brackets.
• `listvar(proc)`:	all names of currently available library procedures.

lres: computes a free resolution of a homogeneous ideal using La Scala's algorithm. It can be used in the same way as `hres`.

maxideal: `maxideal(i)` returns the i–th power of the maximal ideal generated by all ring variables (`maxideal(i)=1` for `i<=0`).

memory: returns statistics concerning the memory management:

- `memory(0)` is the number of active (used) bytes,
- `memory(1)` is the number of bytes allocated from the operating system,
- `memory(2)` is the maximal number of bytes ever allocated from the operating system during the current Singular session.

To monitor the memory usage during ongoing computations the option `mem` should be set.

minbase: `minbase(I)` returns a minimal set of generators of an ideal, respectively module I, if the input is either homogeneous or if the ordering is local.

minor: `minor(M,r)` returns the set of all minors (=subdeterminants) of the given size r of a matrix M. The optional third argument must be a standard basis. If a third argument is given, the computations will be performed modulo that ideal.

minres: `minres(L)` minimizes a free resolution L of an ideal or module.

modulo: `modulo(h1,h2)` returns generators of the kernel of the map $R^k \to R^\ell/h_2$ induced by h_1; k is the number of generators of h_1, that is, it represents $h_1/(h_1 \cap h_2) \cong (h_1 + h_2)/h_2$ where h_1 and h_2 are considered as submodules of the same free module R^ℓ ($\ell = 1$ for ideals).

monitor: `monitor("xxx.txt", "io")` controls the recording of all user input and/or programme output into the file `xxx.txt`. The second argument describes that to log: `"i"` means input, `"o"` means output, `"io"` for both. The default for the second argument is `"i"`. Each `monitor` command closes a previous monitor file and opens the file given by the first string expression. `monitor ("")` turns off recording.

mpresmat: `mpresmat(I,n)` computes the multipolynomial resultant matrix of the ideal I. It uses the sparse resultant matrix method of Gelfand, Kapranov and Zelevinsky (second parameter = 0) or the resultant matrix method of Macaulay (second parameter = 1).
When using the resultant matrix method of Macaulay the input system must be homogeneous. The number of elements in the input system must be the number of variables in the basering, plus one.

mres: computes a minimal free resolution of an ideal or module M by the standard basis method. More precisely, let $A = \mathtt{matrix}(M)$, then `mres(M,k)` computes a free resolution of $\mathrm{Coker}(A) = F_0/M$ [4]

$$\ldots \to F_2 \xrightarrow{A_2} F_1 \xrightarrow{A_1} F_0 \to F_0/M \to 0\,,$$

where the columns of the matrix A_1 are a minimal set of generators of M if the basering is local or if M is homogeneous. If k is not zero then the computation stops after k steps and returns a list of modules $M_i = \mathtt{module}(A_i), i = 1 \ldots k$.
`mres(M,0)` returns a resolution consisting of, at most, $n+2$ modules, where n is the number of variables of the basering. Let `list L=mres(M,0);` then `L[1]` consists of a minimal set of generators of the input, `L[2]` consists of a minimal set of generators for the first syzygy module of `L[1]`, etcetera.

mstd: `mstd(I)` returns a list whose first entry is a standard basis for the ideal, respectively module I. If the monomial ordering is global then the second entry is both a generating set for the ideal, respectively module, and a subset of the standard basis. If, additionally, the input is homogeneous then the second entry is a minimal generating set for I.

mult: `mult(I)` computes the multiplicity of the monomial ideal, respectively module I, generated by the leading monomials of the input. If the

[4] To obtain a minimal free resolution of F_0/M, use `mres(prune(M),0)`.

input is a standard basis of an ideal, respectively module, with respect to a local degree ordering then it returns the multiplicity of this ideal, respectively module, (in the sense of Samuel, with respect to the maximal ideal of the basering).

nameof: returns the name of an expression as string.

names: `names();` returns the names of all user–defined variables, which are ring independent (this includes the names of procedures), `names(R);` returns the names belonging to the ring R.

ncols: `ncols(M)` returns the number of columns of a matrix M or an intmat or the number of given generators of the ideal, including zeros.

npars: `npars(R)` returns the number of parameters of a ring R.

nres: computes a free resolution of an ideal or module M, which is minimized from the second module onwards (by the standard basis method).

nrows: `nrows(M)` returns the number of rows of a matrix M, an intmat or an intvec, respectively the minimal rank, of a free module in which the given module or vector lives (the index of the last non–zero component).

nvars: `nvars(R)` returns the number of variables of a ring R.

open: `open(L)` opens the link L.

option: lists all defined options. `option(option_name)` sets an option. To disable an option, use the prefix `no`. The state of all options to an intvec is dumped by `option(get)`. The state of all options from an intvec (produced by `option(get)`) is restored by `option(set,intvec_expression)`.

The following options are used to manipulate the behaviour of computations and act like boolean switches. Use the prefix `no` to disable an option. Notice that some options are ring dependent and reset to their default values on a change of the current basering:

`none:` turns off all options.

`returnSB:` the functions `syz`, `intersect`, `quotient`, `modulo` return a standard base instead of a generating set if `returnSB` is set. This option should not be used for `lift`.

`fastHC:` tries to the find the highest corner of the staircase as fast as possible during a standard basis computation (only used for local orderings).

`intStrategy:` avoids division of coefficients during standard basis computations. This option is ring dependent. By default, it is set for rings with characteristic 0 and not set for all other rings.

`minRes:` special (additional) minimizing during computations (`res`, `mres`), assumes homogeneous case and degree-compatible ordering.

`notRegularity:` disables the regularity bound for `nres` and `mres`.

`notSugar:` disables the sugar strategy during standard basis computation.

`notBuckets:` disables the bucket representation of polynomials during standard basis computations. This option usually decreases the memory usage but increases the computation time. It should only be set for memory critical standard basis computations.

`prot:` shows protocol information indicating the progress during the following computations: `facstd`, `fglm`, `groebner`, `lres`, `mres`, `minres`, `mstd`, `res`, `sres`, `std`, `stdfglm`, `stdhilb`, `syz`.

`redSB:` computes reduced standard bases S (up to normalization) in any standard basis computation in rings with global monomial orderings. Warning: since, for efficiency reasons, SINGULAR prefers to compute with integers, the leading coefficients are not necessarily 1. Use `simplify(S,1)` to obtain a reduced standard basis.

`redTail:` reduction of the tails of polynomials during standard basis computations. This option is ring dependent. By default, it is set for rings with global degree orderings and not set for all other rings.

`redThrough:` for inhomogeneous input, polynomial reductions during standard basis computations are never postponed, but always finished through. This option is ring dependent. By default, it is set for rings with global degree orderings and not set for all other rings.

`sugarCrit:` uses criteria similar to the homogeneous case to keep more useless pairs.

`weightM:` automatically computes suitable weights for the weighted ecart and the weighted sugar method.

ord: `ord(f)` returns the (weighted) degree of the initial term of the polynomial or a vector f; the weights are the weights used for the first block of the ring ordering. `ord(0)` is `-1`.

ordstr: `ordstr(R)` returns the description of the monomial ordering of the ring R as string.

par: `par(n)` returns the n–th parameter of the basering.

pardeg: `pardeg(p)` returns the degree of a number p considered as a polynomial in the ring parameters.

parstr: `parstr(R)` returns the list of parameters of the ring R as a string. `parstr(R,n)` returns the name of the n–th parameter. If the ring name is omitted, the basering is used, thus `parstr(n)` is equivalent to `parstr(basering,n)`.

preimage: `preimage(R,phi,I)` returns the preimage of the ideal I under the map `phi`. The second argument `phi` has to be a map from the basering to the given ring R (or an ideal defining such a map), and the ideal has to be an ideal in the given ring R. To compute the kernel of a map, the preimage of the zero–ideal has to be computed.

prime: `prime(n)` returns the largest prime smaller or equal to n; returns 2 for all arguments smaller than 3 and 32003 for arguments ≥ 32004.

print: `print(E)` prints the expression E to the terminal and has no return value. `print(E,F)` prints the expression E in a special format. The format string F determines which format to use to generate the output:

`"betti"` The Betti numbers are printed in a matrix–like format where the entry d in row i and column j is the minimal number of generators in degree $i+j$ of the j–th syzygy module of R^n/M

(the 0–th, respectively 1–st, syzygy module of R^n/M are R^n, respectively M).

`"%s"` returns `string(` expression `)`.

`"%2s"` similar to `"%s"`, except that newlines are inserted after every comma and at the end.

`"%l"` similar to `"%s"`, except that each object is embraced by its type such that it can be directly used for "cutting and pasting"

`"%2l"` similar to `"%l"`, except that newlines are inserted after every comma and at the end.

`"%;"` returns the string equivalent to typing `expression;`.

`"%t"` returns the string equivalent to typing `type expression;`.

`"%p"` returns the string equivalent to typing `print(expression);`.

`"%b"` returns the string equivalent to typing `print(expression, "betti");`.

prune: `prune(M)` returns the module M minimally embedded in a free module such that the corresponding factor modules are isomorphic.

qhweight: `qhweight(I)` computes the weight vector of the variables for a quasihomogeneous ideal I. If the input is not weighted homogeneous, an intvec of zeros is returned.

quote: prevents expressions from evaluation. Used only in connection with write to MPfile links, prevents evaluation of an expression before sending it to another SINGULAR process. Within a quoted expression, the quote can be "undone" by an `eval` (that is, each `eval` "undoes" the effect of exactly one quote).

quotient: computes the ideal quotient, respectively module quotient. Let R be the basering, I, J ideals in R, and M, N modules in R^n. Then `quotient(I,J)`$= \{a \in R \mid aJ \subset I\}$, `quotient(M,J)`$= \{b \in R^n \mid bJ \subset M\}$ and `quotient(M,N)`$= \{a \in R \mid aN \subset M\}$.

random: `random(a,b)` returns a random integer between the integer a and the integer b. `random(a,b,c)` returns a $b \times c$ random intmat. The absolute value of the entries of the matrix is smaller than or equal to the integer a.

read: `read(L)` reads data from a link.
For ASCII links, the content of the entire file is returned as one string. If the ASCII link is the empty string, `read` reads from standard input. For MP links, one expression is read from the link and returned after evaluation. For MPtcp links the `read` command blocks as long as there is no data to be read from the link. The `status` command can be used to check whether or not there is data to be read. For DBM links, a `read` with one argument returns the value of the next entry in the data base, and a `read` with two arguments returns the value to the key given as the second argument from the data base.

reduce: `reduce(I,J)` reduces a polynomial, vector, ideal or module I to its normal form with respect to an ideal or module J represented by a standard basis.[5] Returns 0 if and only if the polynomial (respectively vector, ideal, module) is an element (respectively subideal, submodule) of the ideal (respectively module). `reduce(I,J,1)` does no tail reduction. `reduce(I,J,U)` reduces $U^{-1}I$ modulo J. This works only for zero dimensional ideals (respectively modules) J and gives a reduced normal form; U has to be a diagonal matrix with units on the diagonal.

regularity: `regularity(L)` computes the regularity of a homogeneous ideal, respectively module, from a minimal resolution L.
Let $0 \to \bigoplus_a K[x]e_{a,n} \to \ldots \to \bigoplus_a K[x]e_{a,0} \to I \to 0$ be a minimal resolution of I considered with homogeneous maps of degree 0. The regularity is the smallest number s with the property $\deg(e_{a,i}) \leq s + i$ for all i.
If the input to the commands `res` and `mres` is homogeneous, then the regularity is computed and used as a degree bound during the computation unless `option(notRegularity);` is given.

repart: returns the real part of a number from a complex ground field, returns its argument otherwise.

res: `res(M,k)` computes a (possibly minimal) free resolution of an ideal or module M using a heuristically chosen method. The second argument k specifies the length of the resolution. If it is not positive then k is assumed to be the number of variables of the basering. If a third argument is given, the returned resolution is minimized.

reservedName: `reservedName()` prints a list of all reserved identifiers, `reservedName("xyz")` tests whether the string xyz is a reserved identifier.

resultant: `resultant(f,g,x)` computes the resultant of f and g with respect to the variable x.

rvar: `rvar(x)` returns the number of the variable x if the name is a ring variable of the basering or if the string is the name of a ring variable of the basering; returns 0 if not.

setring: `setring(S)` changes the basering to the (already defined) ring S.

simplify: `simplify(f,n)` returns the "simplified" first argument f depending on the simplification rule given by n. The simplification rules are sums of the following basic rules:

`1`	normalize (make leading coefficients 1),
`2`	erase zero generators (respectively columns),
`4`	keep only the first one of identical generators (respectively columns),
`8`	keep only the first one of generators (respectively columns), which differ only by a factor in the ground field,
`16`	keep only those generators (respectively columns) whose leading monomials differ,
`32`	keep only those generators (respectively columns) whose leading monomials are not divisible by other ones.

[5] If J is not a standard basis, then a warning is displayed (the result has in general no invariant meaning).

size: depends on the type of argument: `size(M)` for an ideal or module M returns the number of (non–zero) generators. For a string, intvec, list or resolution it returns the length, that is, the number of characters, entries or elements. For a polynomial or vector it returns the number of monomials. For a matrix or intmat it returns the number of entries. For a ring it returns the number of elements in the ground field (for $\mathbb{Z}/p$ and algebraic extensions) or -1.

sortvec: `sortvec(I)` computes the permutation v, which orders the ideal, respectively module, I by its leading monomials, starting with the smallest, that is, $I(v[i]) < I(v[i+1])$ for all i if the leading monomials of the generators of I are different.

sres: computes a free resolution of an ideal or module with Schreyer's method. The ideal, respectively module, has to be a standard basis. More precisely, let M be given by a standard basis and $A_1 =$`matrix`(M). Then `sres(M,k)` computes a free resolution of $\mathrm{Coker}(A_1) = F_0/M$

$$\ldots \to F_2 \xrightarrow{A_2} F_1 \xrightarrow{A_1} F_0 \to F_0/M \to 0\,.$$

If the int expression k is not zero then the computation stops after k steps and returns a list of modules (given by standard bases), $M_i =$`module`(A_i), $i = 1 \ldots k$. `sres(M,0)` returns a list of n modules, where n is the number of variables of the basering. Even if `sres` does not compute a minimal resolution, the `betti` command gives the true betti numbers! In many cases of interest `sres` is much faster than any other known method. Let `list L=sres(M,0);` then `L[1]=M` is identical to the input, `L[2]` is a standard basis with respect to the Schreyer ordering of the first syzygy module of `L[1]`, etc. ($L[i] = M_i$ in the notations from above.)

status: returns the status of the link as asked for by the second argument. If a third argument is given, the result of the comparison to the status string is returned: `(status(l,s1)==s2)` is equivalent to `status(l,s1,s2)`. If a fourth integer argument (say, `i`) is given and if `status(l,s1,s2)` yields 0, then the execution of the current process is suspended (the process is put to "sleep") for approximately i microseconds, and afterwards the result of another call to `status(l,s1,s2)` is returned. The latter is useful for "polling" the `read` status of MPtcp links such that busy loops are avoided. Note that on some systems, the minimum time for a process to be put to sleep is one second. The following string expressions are allowed:

- `"name"` the name string given by the definition of the link (usually the filename)
- `"type"` returns `"ASCII"`, `"MPfile"`, `"MPtcp"` or `"DBM"`
- `"open"` returns `"yes"` or `"no"`
- `"openread"` returns `"yes"` or `"no"`
- `"openwrite"` returns `"yes"` or `"no"`

"read"	returns "ready" or "not ready"
"write"	returns "ready" or "not ready"
"mode"	(depending on the type of the link and its status) "","w","a","r" or "rw"

std: **std(I)** returns a standard basis of the ideal or module I with respect to the monomial ordering of the basering. Use an optional second argument of type intvec as Hilbert series if the ideal, respectively module, is homogeneous (Hilbert driven standard basis computation). If the ideal is quasihomogeneous with respect to weights w and if the Hilbert series is computed w.r.t. to these weights, then use w as third argument. Use an optional second argument of type poly, respectively vector, to construct the standard basis from an already computed one (given as the first argument) and one additional generator (the second argument). For global orderings, use the `groebner` command instead, which heuristically chooses the "best" algorithm to compute a Gröbner basis. To view the progress of computations, use `option(prot)`.

stdfglm: `stdfglm(I)` computes the standard basis of the 0–dimensional ideal I in the basering via `fglm` (from the ordering given as the second argument to the ordering of the basering). If no second argument is given, "dp" is used.

stdhilb: `stdhilb(I)` computes a standard basis of the ideal I in the basering, via a Hilbert driven standard basis computation. It contains the computation of the Hilbert function of the homogenized ideal I^h.

subst: `subst(f,x,m)` substitutes the ring variable x by the term (a polynomial of length at most 1) m.
Use `map` for substitutions by polynomials.

syz: `syz(I)` computes the first syzygy (that is, the module of relations of the given generators) of the ideal, respectively module, I.

trace: `trace(A)` returns the trace of the matrix A.

transpose: `transpose(A)` transposes the matrix A.

type: prints the name, level, type and value of a variable. To display the value of an expression, it is sufficient to type the expression followed by `;`.

typeof: returns the type of an expression as string. Possible types are:
"ideal", "int", "intmat", "intvec", "list", "map", "matrix", "module", "number", "none", "poly", "proc", "qring", "resolution", "ring", "string", "vector".

uressolve: `uressolve(I,a,b,c)` computes all complex rootsof the zero–dimensional ideal I. Makes either use of the multipolynomial resultant of Macaulay ($a = 1$), which works only for homogeneous ideals, or uses the sparse resultant of Gelfand, Kapranov and Zelevinsky ($a = 0$).
The third argument b defines the precision of the fractional part if the ground field is the field of rational numbers, otherwise it will be ignored.
The fourth argument c (can be 0, 1 or 2) gives the number of extra runs of Laguerre's algorithm, leading to better results.

vandermonde: `vandermonde(p,v,d)` computes the (unique) polynomial of degree d with prescribed values $v_1, \dots, v_N$ at the points

$$p_0 = (p_{0,1}, \dots, p_{0,n}), \dots, p_{N-1} = (p_{N-1,1}, \dots, p_{N-1,n}),$$

where n denotes the number of ring variables and $N := (d+1)^n$.
The returned polynomial is $\sum c_{\alpha_1 \dots \alpha_n} \cdot x_1^{\alpha_1} \cdot \dots \cdot x_n^{\alpha_n}$, where the coefficients $c_{\alpha_1 \dots \alpha_n}$ are the solution of the (transposed) Vandermonde system of linear equations

$$\sum_{\alpha_1 + \dots + \alpha_n \leq d} c_{\alpha_1 \dots \alpha_n} \cdot p_{k-1,1}^{\alpha_1} \cdot \dots \cdot p_{k-1,n}^{\alpha_n} = v_k, \quad k = 1, \dots, N\,.$$

The ground field has to be the field of rational numbers.

var: `var(n)` returns the n–th ring variable.

varstr: `varstr(R)` returns the list of the names of the ring variables as a string: `varstr(R,n)` returns the name of the n–th ring variable. If the ring name is omitted, the basering is used, thus `varstr(n)` is equivalent to `varstr(basering,n)`.

vdim: `vdim(I)` computes the vector space dimension of the ring, respectively free module, modulo the ideal, respectively module, I generated by the initial terms of the given generators. If the generators form a standard basis, this is the same as the vector space dimension of the ring, respectively free module, modulo the ideal, respectively module I. If the ideal, respectively module, is not zero–dimensional, -1 is returned.

wedge: `wedge(M,n)` computes the n–th exterior power of the matrix M.

weight: `weight(I)` computes an "optimal" weight vector for an ideal, respectively module I, which may be used as weight vector for the variables in order to speed up the standard basis algorithm. If the input is weighted homogeneous, a weight vector for which the input is weighted homogeneous is found.

write: writes data to a link.
If the link is of type `ASCII`, all expressions are converted to strings (and separated by a newline character) before they are written. As a consequence, only such values, which can be converted to a string can be written to an `ASCII` link. For MP links, ring–dependent expressions are written together with a ring description. To prevent an evaluation of the expression before it is written, the `quote` command (possibly together with `eval`) can be used. A `write` call blocks (that is, does not return to the prompt), as long as a MPtcp link is not ready for writing. For DBM links, `write` with three arguments inserts the first string as key and the second string as value into the DBM data base. Called with two arguments, it deletes the entry with the key specified by the string from the data base.

B.6 Control Structures

A sequence of commands surrounded by curly brackets is a so–called *block*. Blocks are used in SINGULAR to define procedures and to collect commands belonging to `if`, `else`, `for` and `while` statements and to the `example` part in libraries. Even if the sequence of statements consists of only a single command it has to be surrounded by curly brackets! Variables, which are defined inside a block, are local to that block. Note that there is no ending semicolon at the end of the block.

break: leaves the innermost `for` or `while` block.

breakpoint: sets a breakpoint at the beginning of the specified procedure or at the given line. **Note:** Line number 1 is the first line of a library (for procedures from libraries), respectively the line with the `{`. A line number of -1 removes all breakpoints from that procedure.

continue: skips the rest of the innermost `for` or `while` loop und jumps to the beginning of the block. This command is only valid inside a `for` or a `while` construction.

else: executes the false block if the boolean expression of the `if` statement is false. This command is only valid in combination with an `if` command.

export: converts a local variable of a procedure to a global one. Objects defined in a ring are not automatically exported when exporting the ring (use `keepring` instead).

for: repetitive, conditional execution of a command block. In `for(i=1; i<7;i++)`, the init command `i=1` is executed first. Then the boolean expression `i<7` is evaluated. If its value is *TRUE* the block is executed, otherwise the `for` statement is complete. After each execution of the block, the iterate command `i++` is executed and the boolean expression is evaluated. This is repeated until the boolean expression evaluates to *FALSE*. The command `break;` leaves the innermost `for` construct.

if: executes true block if the boolean condition is true. If the `if` statement is followed by an `else` statement and the boolean condition is false, then false block is executed.

keepring: moves the specified ring to the next (upper) level. This command can only be used inside of procedures and it should be the last command before the `return` statement. There it provides the possibility to keep a ring, which is local to the procedure (and its objects) accessible after the procedure ended without making the ring global.

quit: quits SINGULAR; works also from inside a procedure. The commands `quit` and `exit` are synonymous.

return: returns the result(s) of a procedure and can only be used inside a procedure. Note that the brackets are required even if no return value is given, `return()`.

while: repetitive, conditional execution of a block. In `while(i<7)`, the boolean expression `i<7` is evaluated and if its value is *TRUE*, the block is executed. This is repeated until the boolean expression evaluates to *FALSE*. The command `break` leaves the innermost `while` construction.

~(break point): sets a break point. Whenever SINGULAR reaches the command `~;` in a sequence of commands it prompts for input. The user may now input lines of SINGULAR commands. The line length cannot exceed 80 characters. SINGULAR proceeds with the execution of the command following `~;` as soon as it receives an empty line.

B.7 System Variables

System variables can be used to modify the default behaviour of SINGULAR.

degBound: The standard basis computation is stopped if the (weighted) total degree exceeds `degBound`. `degBound` should not be used for a global ordering with inhomogeneous input. Reset this bound by setting `degBound` to 0.

echo: Input is echoed if `echo >= voice`. `echo` is a local setting for a procedure and defaulted to 0. `echo` does not affect the output of commands.

minpoly: describes the coefficient field of the current basering as an algebraic extension with the minimal polynomial equal to `minpoly`. Setting the `minpoly` should be the first command after defining the ring.
The minimal polynomial has to be specified in the syntax of a polynomial. Its variable is not one of the ring variables, but the algebraic element, which is adjoined to the field. Algebraic extensions in SINGULAR are only possible over the rational numbers or over $\mathbb{Z}/p$, p a prime number.
SINGULAR does not check whether the given polynomial is irreducible! It can be checked in advance with the function `factorize`.

multBound: The standard basis computation is stopped if the ideal is zero–dimensional in a ring with local ordering and its multiplicity (`mult`) is lower than `multBound`. Reset this bound by setting `multBound` to 0.

noether: The standard basis computation in local rings cuts off all monomials above (in the sense of the monomial ordering) the monomial `noether` during the computation. Reset `noether` by setting `noether` to 0.

printlevel: sets the debug level for `dbprint`. If `printlevel >= voice` then `dbprint` is equivalent to `print`, otherwise nothing is printed.

short: the output of monomials is done in the short manner, if `short` is non–zero. A C–like notation is used, if short is zero. Both notations may be used as input. The default depends on the names of the ring variables (0 if there are names of variables longer than one character, 1 otherwise). Every change of the basering sets `short` to the previous value for that ring. In other words, the value of the variable `short` is “ring-local”.

TRACE: sets level of debugging.

`TRACE=0` no debugging messages are printed.
`TRACE=1` messages about entering and leaving of procedures are displayed.

`TRACE=3` messages about entering and leaving of procedures together with line numbers are displayed.
`TRACE=4` each line is echoed and the interpretation of commands in this line is suspended until the user presses `RETURN`.
`TRACE` is defaulted to 0. It does not affect the output of commands.

voice: shows the nesting level of procedures.

B.8 Libraries

```
LIBRARY: ainvar.lib      Invariant Rings of the Additive Group (see also [92])
AUTHORS: Gerhard Pfister,     pfister@mathematik.uni-kl.de,
         Gert-Martin Greuel,  greuel@mathematik.uni-kl.de

PROCEDURES:
  invariantRing(m..);     compute ring of invariants of (K,+)-action given by m
  derivate(m,f);          derivation of f with respect to the vectorfield m
  actionIsProper(m);      tests whether action defined by m is proper
  reduction(p,I);         SAGBI reduction of p in the subring generated by I
  completeReduction();    complete SAGBI reduction
  localInvar(m,p..);      invariant polynomial under m computed from p,...
  furtherInvar(m..);      compute further invariants of m from the given ones
  sortier(id);            sorts generators of id by increasing leading terms
```

```
LIBRARY: algebra.lib     Compute with Algebras and Algebra Maps
AUTHORS: Gert-Martin Greuel,     greuel@mathematik.uni-kl.de,
         Agnes Eileen Heydtmann, agnes@math.uni-sb.de,
         Gerhard Pfister,        pfister@mathematik.uni-kl.de

PROCEDURES:
 algebra_containment(); query of algebra containment
 module_containment();  query of module containment over a subalgebra
 inSubring(p,I);        test whether poly p is in subring generated by I
 algDependent(I);       computes algebraic relations between generators of I
 alg_kernel(phi,R);     computes the kernel of the ring map phi
 is_injective(phi,R);   test for injectivity of ring map phi
 is_surjective(phi,R);  test for surjectivity of ring map phi
 is_bijective(phi,R);   test for bijectivity of ring map phi
 noetherNormal(id);     Noether normalization of ideal id
 mapIsFinite(R,phi,I);  query for finiteness of map phi: R -> basering/I
```

```
LIBRARY: brnoeth.lib     Brill-Noether Algorithm, Weierstraß-SG and AG-codes
AUTHORS: Jose Ignacio Farran Martin, ignfar@eis.uva.es
         Christoph Lossen,           lossen@mathematik.uni-kl.de

OVERVIEW:
```

Implementation of the Brill–Noether algorithm for solving the Riemann–Roch problem and applications in Algebraic Geometry codes. The computation of Weierstraß semigroups is also implemented. The procedures are intended only for plane (singular) curves defined over a prime field of positive characteristic.

```
MAIN PROCEDURES:
 Adj_div(f);              computes the conductor of a curve
 NSplaces(h,A);           computes non-singular places up to given degree
 BrillNoether(D,C);       computes a vector space basis of the linear system L(D)
 Weierstrass(P,m,C);      computes the Weierstraß semigroup of C at P up to m
 extcurve(d,C);           extends the curve C to an extension of degree d
 AGcode_L(G,D,E);         computes the evaluation AG code with divisors G and D
 AGcode_Omega(G,D,E);     computes the residual AG code with divisors G and D
 prepSV(G,D,F,E);         preprocessing for the basic decoding algorithm
 decodeSV(y,K);           decoding of a word with the basic decoding algorithm

AUXILIARY PROCEDURES:
 closed_points(I);        computes the zero-set of a zero-dim. ideal in 2 vars
 dual_code(C);            computes the dual code
 sys_code(C);             computes an equivalent systematic code
 permute_L(L,P);          applies a permutation to a list


LIBRARY: classify.lib   Arnold Classifier of Singularities
AUTHOR:  Kai Krüger,     krueger@mathematik.uni-kl.de
         Corina Baciu,   baciu@mathematik.uni-kl.de

OVERVIEW:
```

A library for classifying isolated hypersurface singularities w.r.t. right equivalence, based on the determinator of singularities by V.I. Arnold.

```
PROCEDURES:
 basicinvariants(f); computes Milnor number, determinacy-bd. and crk of f
 classify(f);        normal form of poly f determined with Arnold's method
 corank(f);          computes the corank of f (i.e. of the Hessian of f)
 Hcode(v);           coding of intvec v according to the number repetitions
 init_debug([n]);    print trace and debugging information depending on int n
 internalfunctions(); display names of internal procedures of this library
 milnorcode(f[,e]);  Hilbert poly of [e-th] Milnor algebra coded with Hcode
 morsesplit(f);      residual part of f after applying the splitting lemma
 quickclass(f)       normal form of f determined by invariants (milnorcode)
 singularity(s,[]);  normal form of singularity given by its name s and index
 swap(a,b);          returns b,a
 A_L(s/f)            shortcut for quickclass(f) or normalform(s)
 normalform(s);      normal form of singularity given by its name s
 debug_log(lev,[]);  print trace and debugging information w.r.t level>@DeBug
           (parameters in square brackets [] are optional)


LIBRARY: deform.lib     Miniversal Deformation of Singularities and Modules
AUTHOR:  Bernd Martin,  martin@math.tu-cottbus.de

PROCEDURES:
 versal(Fo[,d,any]);          miniversal deformation of isolated singularity Fo
 mod_versal(Mo,I,[,d,any]);   miniversal deformation of module Mo mod ideal I
```

```
 lift_kbase(N,M);            lifting N into standard kbase of M
 lift_rel_kb(N,M[,kbM,p]);   relative lifting N into a kbase of M
 kill_rings(["prefix"]);     kills the exported rings from above


LIBRARY: elim.lib         Elimination, Saturation and Blowing up

PROCEDURES:
 blowup0(j[,s1,s2]);      create presentation of blownup ring of ideal j
 elim(id,n,m);            variable n..m eliminated from id (ideal/module)
 elim1(id,p);             p=product of vars to be eliminated from id
 nselect(id,n[,m]);       select generators not containing nth [..m-th] variable
 sat(id,j);               saturated quotient of ideal/module id by ideal j
 select(id,n[,m]);        select generators containing all variables n...m
 select1(id,n[,m]);       select generators containing one variable n...m


LIBRARY: equising.lib    Equisingularity Stratum of a Family of Plane Curves
AUTHOR:  Andrea Mindnich,  mindnich@mathematik.uni-kl.de

PROCEDURES:
 esStratum(F[,m]);        computes the equisingularity stratum of the family F
 isEquising(F[,m]);       tests if a given deformation is equisingular


LIBRARY: finvar.lib      Invariant Rings of Finite Groups
AUTHOR:  Agnes E. Heydtmann,  agnes@math.uni-sb.de

OVERVIEW:
```

A library for computing polynomial invariants of finite matrix groups and generators of related varieties. The algorithms are based on B. Sturmfels, G. Kemper and W. Decker et al.

```
MAIN PROCEDURES:
 invariant_ring();                generators of the invariant ring (i.r.)
 invariant_ring_random();         generators of the i.r., randomized alg.
 primary_invariants();             primary invariants (p.i.)
 primary_invariants_random();     primary invariants, randomized alg.

AUXILIARY PROCEDURES:
 cyclotomic();                    cyclotomic polynomial
 group_reynolds();                finite group and Reynolds operator (R.o.)
 molien();                        Molien series (M.s.)
 reynolds_molien();               Reynolds operator and Molien series
 partial_molien();                partial expansion of Molien series
 evaluate_reynolds();             image under the Reynolds operator
 invariant_basis();               basis of homogeneous invars of a degree
 invariant_basis_reynolds();      as invariant˙basis(), with R.o.
 primary_char0();                 primary invariants in char 0
 primary_charp();                 primary invariant in char p
 primary_char0_no_molien();       p.i., char 0, without Molien series
 primary_charp_no_molien();       p.i., char p, without Molien series
 primary_charp_without();         p.i., char p, without R.o. or Molien series
```

```
primary_char0_random();             primary invariants in char 0, randomized
primary_charp_random();             primary invariants in char p, randomized
primary_char0_no_molien_random();   p.i., char 0, without M.s., randomized
primary_charp_no_molien_random();   p.i., char p, without M.s., randomized
primary_charp_without_random();     p.i., char p, without R.o. or M.s., random.
power_products();                   exponents for power products
secondary_char0();                  secondary (s.i.) invariants in char 0
secondary_charp();                  secondary invariants in char p
secondary_no_molien();              secondary invariants, without M.s.
secondary_and_irreducibles_no_molien(); s.i. & irreducible s.i., without M.s.
secondary_not_cohen_macaulay();     s.i. when invariant ring not CM
orbit_variety();                    ideal of the orbit variety
relative_orbit_variety();           ideal of a relative orbit variety
image_of_variety();                 ideal of the image of a variety
```

```
LIBRARY: gaussman.lib    Gauß-Manin Connection of a Singularity
AUTHOR:  Mathias Schulze,   mschulze@mathematik.uni-kl.de
```

OVERVIEW:

A library to compute invariants related to the Gauß–Manin connection of an isolated hypersurface singularity.

```
PROCEDURES:
 monodromy(f[,...]);     monodromy matrix of f, spectrum of monodromy of f
 vfiltration(f[,...]);   V-filtration of f on H''/H', singularity spectrum of f
 vfiltjacalg(...);       V-filtration on Jacobian algebra
 gamma(...);             C. Hertling's gamma invariant
```

```
LIBRARY: general.lib     Elementary Computations of General Type
```

```
PROCEDURES:
 A_Z("a",n);             string a,b,... of n comma separated letters
 ASCII([n,m]);           string of printable ASCII characters (number n to m)
 binomial(n,m[,../..]);  n choose m (type int), [type string/type number]
 cr_roots(n,m);          complex and real (positive) m-th roots of number n
 chebychev(n);           n-th Chebychev polynomial
 deleteSublist(iv,l);    delete entries given by iv from list l
 factorH(p);             factorizes with good choice of principal variable
 factorial(n[,../..]);   n factorial (=n!) (type int), [type string/number]
 fibonacci(n[,p]);       n-th Fibonacci number [char p]
 kmemory([n[,v]]);       active [allocated] memory in kilobyte
 killall();              kill all user-defined variables
 number_e(n);            compute exp(1) up to n decimal digits
 number_pi(n);           compute pi (area of unit circle) up to n digits
 primecoeffs(J[,q]);     prime factors ≤ min(p,32003) of coeffs of J
 primefactors(n[,p]);    prime factors ≤ min(p,32003) of n
 primes(n,m);            intvec of primes p, n ≤ p ≤ m
 product(../..[,v]);     multiply components of vector/ideal/...[indices v]
 sort(ideal/module);     sort generators according to monomial ordering
 sum(vector/id/..[,v]);  add components of vector/ideal/...[with indices v]
 timeStd(i,d);           std(i) if computation finished after d seconds else i
```

`timeFactorize(p,d);`	works as timeStd with factorization
`watchdog(i,cmd);`	only wait for result of command cmd for i seconds
`which(command);`	search for command and return absolute path, if found

```
LIBRARY: graphics.lib    Procedures to use Graphics with Mathematica
AUTHOR:  Christian Gorzel, gorzelc@math.uni-muenster.de
```

`PROCEDURES:`

`staircase(fname,I);`	Mathematica text for displaying staircase of I
`mathinit();`	string for loading Mathematica's ImplicitPlot
`mplot(fname,I[# s]);`	Mathematica text for various plots

```
LIBRARY: hnoether.lib    Hamburger-Noether (Puiseux) Development
AUTHOR:  Martin Lamm,    lamm@mathematik.uni-kl.de
```

`OVERVIEW:`

A library for computing the Hamburger–Noether, respectively Puiseux, development of a plane curve singularity following [34]. The library contains also procedures for computing the (topological) numerical invariants of plane curve singularities.

`MAIN PROCEDURES:`

`HNdevelop(f);`	Hamburger-Noether (H-N) development of f
`develop(f[,n]);`	H-N development of irreducible curves
`reddevelop(f);`	H-N development of reducible curves
`essdevelop(f);`	H-N development of essential branches
`extdevelop(hne,n);`	extension of the H-N development hne of f
`param(hne [,x]);`	a parametrization of f (input=output(develop))
`displayHNE(hne);`	display H-N development as an ideal
`invariants(hne);`	invariants of f, e.g. the characteristic exponents
`multsequence(hne);`	sequence of multiplicities
`displayInvariants(hne);`	display invariants of f
`intersection(hne1,hne2);`	intersection multiplicity of two curves
`displayMultsequence(hne);`	display sequence of multiplicities
`stripHNE(hne);`	reduce amount of memory consumed by hne
`is_irred(f);`	test for irreducibility
`deltaHNE(hne);`	delta-invariant of f, hne=reddevelop(f)
`delta(f);`	delta-invariant of f

`AUXILIARY PROCEDURES:`

`puiseux2generators(m,n);`	convert Puiseux pairs to generators of semigroup
`multiplicities(hne);`	multiplicities of blowed up curves
`separateHNE(hne1,hne2);`	number of quadratic transf. needed for separation
`squarefree(f);`	a squarefree divisor of the poly f
`allsquarefree(f,l);`	the maximal squarefree divisor of the poly f
`further_hn_proc();`	show further procedures useful for interactive use

```
LIBRARY: homolog.lib    Procedures for Homological Algebra
AUTHORS: Gert-Martin Greuel,  greuel@mathematik.uni-kl.de,
         Bernd Martin,        martin@math.tu-cottbus.de

PROCEDURES:
 cup(M);                     ∪ : Ext^1(M',M') × Ext^1() → Ext^2()
 cupproduct(M,N,P,p,q);      ∪ : Ext^p(M',N') × Ext^q(N',P') → Ext^{p+q}(M',P')
 Ext_R(k,M);                 Ext^k(M',R), M module, R basering, M'=coker(M)
 Ext(k,M,N);                 Ext^k(M',N'), M'=coker(M), N'=coker(N)
 Hom(M,N);                   Hom(M',N'), M'=coker(M), N'=coker(N)
 homology(A,B,M,N);          ker(B)/im(A), homology of complex R^k --A--> M' --B--> N'
 kernel(A,M,N);              ker(A:M' → N')  M,N modules, A matrix
 kohom(A,k);                 Hom(R^k,A), A matrix over basering R
 kontrahom(A,k);             Hom(A,R^k), A matrix over basering R


LIBRARY: inout.lib      Printing and Manipulating In- and Output

PROCEDURES:
 allprint(list);        print list if ALLprint is defined, with pause if >0
 lprint(poly/...[,n]);  display poly/... fitting to pagewidth [size n]
 pmat(matrix[,n]);      print form-matrix [first n chars of each column]
 rMacaulay(string);     read Macaulay'1 output and return its Singular format
 show(any);             display any object in a compact format
 showrecursive(id,p);   display id recursively with respect to variables in p
 split(string,n);       split given string into lines of length n
 tab(n);                string of n space tabs
 writelist(...);        write a list into a file and keep the list structure
 pause([prompt]);       stop the computation until user input


LIBRARY: intprog.lib    Integer Programming with Gröbner Basis Methods
AUTHOR:  Christine Theis,  ctheis@math.uni-sb.de

PROCEDURES:
 solve_IP(..);    procedures for solving integer programming problems


LIBRARY: latex.lib      Typesetting of Singular-Objects in LaTeX2e
AUTHOR: Christian Gorzel,  gorzelc@math.uni-muenster.de

PROCEDURES:
 closetex(fnm);         writes closing line for LaTeX-document
 opentex(fnm);          writes header for LaTeX-file fnm
 tex(fnm);              calls LaTeX2e for LaTeX-file fnm
 texdemo([n]);          produces a file explaining the features of this lib
 texfactorize(fnm,f);   creates string in LaTeX-format for factors of poly f
 texmap(fnm,m,r1,r2);   creates string in LaTeX-format for map m: r1 → r2
 texname(fnm,s);        creates string in LaTeX-format for identifier
 texobj(l);             creates string in LaTeX-format for any (basic) type
 texpoly(f,n[,l]);      creates string in LaTeX-format for poly
```

`texproc(fnm,p);`	creates string in LaTeX-format of text from proc p
`texring(fnm,r[,l]);`	creates string in LaTeX-format for ring/qring
`rmx(s);`	removes .aux and .log files of LaTeX-files
`xdvi(s);`	calls xdvi for dvi-files

`LIBRARY: linalg.lib` Algorithmic Linear Algebra
`AUTHORS: Ivor Saynisch, ivs@math.tu-cottbus.de`
`Mathias Schulze, mschulze@mathematik.uni-kl.de`

`PROCEDURES:`

`inverse(A);`	the inverse matrix of A
`inverse_B(A);`	list(matrix Inv,poly p), $\mathrm{Inv} \cdot \mathrm{A} = \mathrm{p} \cdot \mathrm{En}$ (using busadj(A))
`inverse_L(A);`	list(matrix Inv,poly p), $\mathrm{Inv} \cdot \mathrm{A} = \mathrm{p} \cdot \mathrm{En}$ (using lift)
`sym_gauss(A);`	symmetric Gaussian algorithm
`orthogonalize(A);`	Gram-Schmidt orthogonalization
`diag_test(A);`	test whether A can be diagonalized
`busadj(A);`	coeffs of $\mathrm{Adj}(\mathrm{En} \cdot t - \mathrm{A})$ and coeffs of $\det(\mathrm{En} \cdot t - \mathrm{A})$
`charpoly(A,v);`	characteristic polynomial of A (using busadj(A))
`adjoint(A);`	adjoint of A (using busadj(A))
`det_B(A);`	determinant of A (using busadj(A))
`gaussred(A);`	Gaussian reduction: $\mathrm{P} \cdot \mathrm{A} = \mathrm{U} \cdot \mathrm{S}$, S row reduced form of A
`gaussred_pivot(A);`	Gaussian reduction: $\mathrm{P} \cdot \mathrm{A} = \mathrm{U} \cdot \mathrm{S}$, uses row pivoting
`gauss_nf(A);`	Gaussian normal form of A
`mat_rk(A);`	rank of constant matrix A
`U_D_O(A);`	$\mathrm{P} \cdot \mathrm{A} = \mathrm{U} \cdot \mathrm{D} \cdot \mathrm{O}$, P,D,U,O = permutation, diagonal, lower-, upper-triangular matrix
`pos_def(A,i);`	test symmetric matrix for positive definiteness
`jordan(M[,opt]);`	eigenvalues, Jordan block sizes, transformation matrix
`jordanmatrix(l);`	Jordan matrix with eigenvalues, Jordan block sizes
`jordanform(M);`	Jordan normal form of constant square matrix M

`LIBRARY: LLL.lib` Integral LLL-Algorithm (see also [122])
`AUTHOR: Alberto Vigneron-Tenorio, alberto.vigneron@uca.es`
`Alfredo Sanchez-Navarro, alfredo.sanchez@uca.es`

`PROCEDURES:`

`LLL(..);`	Integral LLL-Algorithm

`LIBRARY: matrix.lib` Elementary Matrix Operations

`PROCEDURES:`

`compress(A);`	matrix, zero columns from A deleted
`concat(A1,A2,..);`	matrix, concatenation of matrices A1,A2,...
`diag(p,n);`	matrix, nxn diagonal matrix with entries poly p
`dsum(A1,A2,..);`	matrix, direct sum of matrices A1,A2,...
`flatten(A);`	ideal, generated by entries of matrix A
`genericmat(n,m[,id]);`	generic nxm matrix [entries from id]
`is_complex(c);`	1 if list c is a complex, 0 if not
`outer(A,B);`	matrix, outer product of matrices A and B
`power(A,n);`	matrix/intmat, n-th power of matrix/intmat A

`skewmat(n[,id]);`	generic skew-symmetric nxn matrix [entries from id]
`submat(A,r,c);`	submatrix of A with rows/cols specified by intvec r/c
`symmat(n[,id]);`	generic symmetric nxn matrix [entries from id]
`tensor(A,B);`	matrix, tensor product of matrices A and B
`unitmat(n);`	unit square matrix of size n
`gauss_col(A);`	transform a matrix into col-reduced Gauß normal form
`gauss_row(A);`	transform a matrix into row-reduced Gauß normal form
`addcol(A,c1,p,c2);`	add p · (c1-th col) to c2-th column of matrix A, p poly
`addrow(A,r1,p,r2);`	add p · (r1-th row) to r2-th row of matrix A, p poly
`multcol(A,c,p);`	multiply c-th column of A with poly p
`multrow(A,r,p);`	multiply r-th row of A with poly p
`permcol(A,i,j);`	permute i-th and j-th columns
`permrow(A,i,j);`	permute i-th and j-th rows
`rowred(A[,any]);`	reduction of matrix A with elementary row-operations
`colred(A[,any]);`	reduction of matrix A with elementary col-operations
`rm_unitrow(A);`	remove unit rows and associated columns of A
`rm_unitcol(A);`	remove unit columns and associated rows of A

`LIBRARY: mondromy.lib` Monodromy of an Isolated Hypersurface Singularity
`AUTHOR: Mathias Schulze, mschulze@mathematik.uni-kl.de`

`OVERVIEW:`

A library to compute the monodromy of an isolated hypersurface singularity. It uses an algorithm by Brieskorn [24] to compute a connection matrix of the meromorphic Gauß-Manin connection up to arbitrarily high order, and an algorithm of Gerard and Levelt [69] to transform it to a simple pole.

`PROCEDURES:`

`detadj(U);`	determinant and adjoint matrix of square matrix U
`invunit(u,n);`	series inverse of polynomial u up to order n
`jacoblift(f);`	lifts f^κ in jacob(f) with minimal κ
`monodromyB(f[,opt]);`	monodromy of isolated hypersurface singularity f
`H2basis(f);`	basis of Brieskorn lattice H''

`LIBRARY: mprimdec.lib` Procedures for Primary Decomposition of Modules
`AUTHORS: Alexander Dreyer, adreyer@web.de`

`REMARK:`

These procedures are implemented to be used in characteristic 0. They also work in positive characteristic $\gg 0$. In small characteristic and for algebraic extensions, the procedures via Gianni, Trager, Zacharias may not terminate.

`PROCEDURES:`

`separator(l);`	computes a list of separators of prime ideals
`PrimdecA(N[,i]);`	(not necessarily minimal) primary decomposition via Shimoyama/Yokoyama (suggested by Gräbe)
`PrimdecB(N,p);`	(not necessarily minimal) primary decomposition for pseudo-primary ideals
`modDec(N[,i]);`	minimal primary decomposition via Shimoyama/Yokoyama (suggested by Gräbe)

```
 zeroMod(N[,check]);         minimal zero-dimensional primary decomposition
                             via Gianni, Trager and Zacharias
 GTZmod(N[,check]);          minimal primary decomposition
                             via Gianni, Trager and Zacharias
 dec1var(N[,check[,ann]]);   primary decomposition for one variable
 annil(N);                   the annihilator of R^n/N in the basering
 splitting(N[,check[,ann]]); splitting to simpler modules
 primTest(i[,p]);            tests whether i is prime or homogeneous
 preComp(N,check[,ann]);     enhanced version of splitting
 indSet(i);                  lists with varstrings of(in)dependent variables
 GTZopt(N[,check[,ann]]);    a faster version of GTZmod
 zeroOpt(N[,check[,ann]]);   a faster version of zeroMod
 clrSBmod(N);                extracts an minimal SB from a SB
 minSatMod(N,I);             minimal saturation of N w.r.t. I
 specialModulesEqual(N1,N2); checks for equality of standard bases of modules
                               if N1 is contained in N2 or vice versa
 stdModulesEqual(N1,N2);     checks for equality of standard bases
 modulesEqual(N1,N2);        checks for equality of modules
 getData(N,l[,i]);           extracts oldData and computes the remaining data
```

```
LIBRARY: mregular.lib   Castelnuovo-Mumford Regularity of CM-Schemes
AUTHORS: Isabel Bermejo,      ibermejo@ull.es
         Philippe Gimenez,    pgimenez@agt.uva.es
         Gert-Martin Greuel,  greuel@mathematik.uni-kl.de
```

OVERVIEW:

A library for computing the Castelnuovo–Mumford regularity of a subscheme of the projective n-space that DOES NOT require the computation of a minimal graded free resolution of the saturated ideal defining the subscheme. The procedures are based on [20], and [21]. The algorithm assumes the variables to be in Noether position.

PROCEDURES:

```
 reg_CM(id);          regularity of arith. C-M subscheme V(id'sat) of Pn
 reg_curve(id,[,e]);  regularity of projective curve V(id'sat) in Pn
 reg_moncurve(li);    regularity of projective monomial curve defined by li
```

```
LIBRARY: normal.lib   Normalization of Affine Rings
AUTHORS: Gert-Martin Greuel,  greuel@mathematik.uni-kl.de,
         Gerhard Pfister,     pfister@mathematik.uni-kl.de
```

PROCEDURES:

```
 normal(I[,"wd"]);   computes the normalization of basering/I, respectively
                       the normalization of basering/I and the delta-invariant
 HomJJ(L);           presentation of End'R(J) as affine ring, L a list
 genus(I);           computes the genus of the projective curve defined by I
```

```
LIBRARY: ntsolve.lib   Real Newton Solving of Polynomial Systems
AUTHORS: Wilfred Pohl,       pohl@mathematik.uni-kl.de
         Dietmar Hillebrand
```

```
PROCEDURES:
 nt_solve(G,ini,[..]);  find one real root of 0-dimensional ideal G
 triMNewton(G,a,[..]);  find one real root for 0-dim triangular system G


LIBRARY: paramet.lib    Parametrization of Varieties
AUTHOR:  Thomas Keilen, keilen@mathematik.uni-kl.de

OVERVIEW:
```

A library to compute parametrizations of algebraic varieties (if possible) with the aid of a normalization, or a primary decomposition, respectively to compute a parametrization of a plane curve singularity with the aid of a Hamburger-Noether expansion.

```
PROCEDURES:
 parametrize(I);        parametrizes a prime ideal via the normalization
 parametrizepd(I);      computes primary dec. and parametrizes components
 parametrizesing(f);    parametrizes an isolated plane curve singularity


LIBRARY: poly.lib         Procedures for Manipulating Polys, Ideals, Modules
AUTHORS: Olaf Bachmann,         obachman@mathematik.uni-kl.de,
         Gert-Martin Greuel,    greuel@mathematik.uni-kl.de,
         Anne Frühbis-Krüger,   anne@mathematik.uni-kl.de

PROCEDURES:
 hilbPoly(I);              Hilbert polynomial of basering/I
 cyclic(int);              ideal of cyclic n-roots
 katsura([i]);             katsura [i] ideal
 freerank(poly/...)        rank of coker(input) if coker is free else -1
 is_homog(poly/...);       int, =1 resp. =0 if input is homogeneous resp. not
 is_zero(poly/...);        int, =1 resp. =0 if coker(input) is 0 resp. not
 lcm(ideal);               lcm of given generators of ideal
 maxcoef(poly/...);        maximal length of coefficient occurring in poly/...
 maxdeg(poly/...);         int/intmat = degree/s of terms of maximal order
 maxdeg1(poly/...);        int = [weighted] maximal degree of input
 mindeg(poly/...);         int/intmat = degree/s of terms of minimal order
 mindeg1(poly/...);        int = [weighted] minimal degree of input
 normalize(poly/...);      normalize poly/... such that leading coefficient is 1
 rad_con(p,I);             check radical containment of poly p in ideal I
 content(f);               content of polynomial/vector f
 numerator(n);             numerator of number n
 denominator(n)            denominator of number n
 mod2id(M,iv);             conversion of a module M to an ideal
 id2mod(i,iv);             conversion inverse to mod2id
 subrInterred(i1,i2,iv);   interred w.r.t. a subset of variables


LIBRARY: presolve.lib    Pre-Solving of Polynomial Equations

PROCEDURES:
 degreepart(id,d1,d2);   elements of id of total degree ≥ d1 and ≤ d2
```

`elimlinearpart(id);`	linear part eliminated from id
`elimpart(id[,n]);`	partial elimination of vars [among first n vars]
`elimpartanyr(i,p);`	factors of p partially eliminated from i in any ring
`fastelim(i,p[..]);`	fast elimination of factors of p from i [options]
`findvars(id[..]);`	ideal of variables occurring in id [more information]
`hilbvec(id[,c,o]);`	intvec of Hilbert series of id [in char c and ord o]
`linearpart(id);`	elements of id of total degree ≤ 1
`tolessvars(id[,]);`	maps id to new basering having only vars occurring in id
`solvelinearpart(id);`	reduced std-basis of linear part of id
`sortandmap(id,s1,s2);`	map to new basering with vars sorted w.r.t. complexity
`sortvars(id[n1,p1..]);`	sort vars w.r.t. complexity in id [different blocks]
`shortid(id,n);`	generators of id having at most n terms
`valvars(id[..]);`	valuation of vars w.r.t. to their complexity in id
`idealSimplify(id);`	eliminates variables which are linear in id
`idealSplit(id,tF,fS);`	radical of the intersection of the ideals =radical(id)

```
LIBRARY: primdec.lib     Primary Decomposition and Radical of Ideals
AUTHORS: Gerhard Pfister,   pfister@mathematik.uni-kl.de (GTZ),
         Wolfram Decker,    decker@math.uni-sb.de        (SY),
         Hans Schönemann,   hannes@mathematik.uni-kl.de  (SY)
```

`OVERVIEW:`

Algorithms for primary decomposition based on ideas of Gianni, Trager and Zacharias [72] (implementation by G. Pfister), respectively based on ideas of Shimoyama and Yokoyama [168] (implementation by W. Decker and H. Schönemann). The procedures are implemented to be used in characteristic 0. They also work in positive characteristic $\gg 0$. In small characteristic and for algebraic extensions, primdecGTZ and minAssGTZ may not terminate, while primdecSY and minAssChar may not give a complete decomposition. Algorithms for the computation of the radical based on the ideas of Krick, Logar [111] and Kemper (implementation by G. Pfister).

`PROCEDURES:`

`Ann(M);`	annihilator of module R^n/M
`primdecGTZ(I);`	complete primary decomposition via Gianni,Trager,Zacharias
`primdecSY(I...);`	complete primary decomposition via Shimoyama-Yokoyama
`minAssGTZ(I);`	the minimal associated primes via Gianni,Trager,Zacharias
`minAssChar(I...);`	the minimal associated primes using characteristic sets
`testPrimary(L,k);`	tests the result of the primary decomposition
`radical(I);`	computes the radical of I via Krick/Logar and Kemper
`radicalEHV(I);`	computes the radical of I via Eisenbud,Huneke,Vasconcelos
`equiRadical(I);`	the radical of the equidimensional part of the ideal I
`prepareAss(I);`	list of radicals of the equidimensional components of I
`equidim(I);`	weak equidimensional decomposition of I
`equidimMax(I);`	equidimensional locus of I
`equidimMaxEHV(I);`	equidimensional locus of I via Eisenbud,Huneke,Vasconcelos
`zerodec(I);`	zero-dimensional decomposition via Monico

```
LIBRARY: primitiv.lib   Computing a Primitive Element
AUTHOR:  Martin Lamm,   lamm@mathematik.uni-kl.de

PROCEDURES:
 primitive(ideal i);    find minimal polynomial for a primitive element
 primitive_extra(i);    find primitive element for two generators
 splitring(f,R[,L]);    define ring extension with name R and switch to it


LIBRARY: qhmoduli.lib   Moduli Spaces of Semi-Quasihomogeneous Singularities
AUTHOR:  Thomas Bayer,  bayert@in.tum.de

OVERVIEW:
```

Compute equations for the moduli space of an isolated semi-quasihomogeneous hypersurface singularity with fixed principal part (based on [84]).

```
PROCEDURES:
 ArnoldAction(f,[G,w]);      induced action of G'f on T'
 ModEqn(f);                  equations of the moduli space for principal part f
 QuotientEquations(G,A,I);   equations of Variety(I)/G w.r.t. action 'A'
 StabEqn(f);                 equations of the stabilizer of f
 StabEqnId(I,w);             equations of the stabilizer of the qhom. ideal I
 StabOrder(f);               order of the stabilizer of f
 UpperMonomials(f,[w]);      upper basis of the Milnor algebra of f
 Max(data);                  maximal integer contained in 'data'
 Min(data);                  minimal integer contained in 'data'
 Table(cmd,i,lb,ub);         list, i-th entry is cmd(i), lb <= i <= ub


LIBRARY: random.lib          Creating Random and Sparse Matrices, Ideals, Polys

PROCEDURES:
 genericid(i[,p,b]);         generic sparse linear combinations of generators of i
 randomid(id,[k,b]);         random linear combinations of generators of id
 randommat(n,m[,id,b]);      nxm matrix of random linear combinations of id
 sparseid(k,u[,o,p,b]);      ideal of k random sparse poly's of degree d
 sparsematrix(n,m,o[,.]);    nxm sparse matrix of polynomials of degree <= o
 sparsemat(n,m[,p,b]);       nxm sparse integer matrix with random coefficients
 sparsepoly(u[,o,p,b]);      random sparse polynomial, terms of degree in [u,o]
 sparsetriag(n,m[,.]);       nxm sparse lower-triang intmat with random coeffs
 triagmatrix(n,m,o[,.]);     nxm sparse lower-triang matrix of poly's of deg <= o
 randomLast(b);              random transformation of the last variable
 randomBinomial(k,u,..);     binomial ideal, k random generators of deg >= u

LIBRARY: reesclos.lib        Procedures to Compute Integral Closure of an Ideal
AUTHOR:  Tobias Hirsch,      hirsch@math.tu-cottbus.de

OVERVIEW:
```

A library to compute the integral closure of an ideal I in a R=k[x(1),...,x(n)] using the Rees–Algebra R[It] of I. It computes the integral closure of R[It] (in the same manner as done in the library 'normal.lib'), which is a graded subalgebra of R[t]. The degree-k-component is the integral closure of the k-th power of I.

```
PROCEDURES:
 ReesAlgebra(I);          computes Rees-Algebra of an ideal I
 normalI(I[,p[,r]]);      computes integral closure of an ideal I using R[It]
```

```
LIBRARY: ring.lib         Manipulating Rings and Maps

PROCEDURES:
 changechar("R",c[,r]);   make a copy R of basering [ring r] with new char c
 changeord("R",o[,r]);    make a copy R of basering [ring r] with new ord o
 changevar("R",v[,r]);    make a copy R of basering [ring r] with new vars v
 defring("R",c,n,v,o);    define a ring R in specified char c, n vars v, ord o
 defrings(n[,p]);         define ring Sn in n vars, char 32003 [p], ord ds
 defringp(n[,p]);         define ring Pn in n vars, char 32003 [p], ord dp
 extendring("R",n,v,o);   extend given ring by n vars v, ord o and name it R
 fetchall(R[,str]);       fetch all objects of ring R to basering
 imapall(R[,str]);        imap all objects of ring R to basering
 mapall(R,i[,str]);       map all objects of ring R via ideal i to basering
 ord_test(R);             test whether ordering of R is global, local or mixed
 ringtensor("R",s,t,..);  create ring R, tensor product of rings s,t,...
 ringweights(r);          intvec of weights of ring variables of ring r
```

```
LIBRARY: rinvar.lib       Invariant Rings of Reductive Groups
AUTHOR:  Thomas Bayer,    tbayer@in.tum.de

OVERVIEW:
```

Implementation based on Derksen's algorithm. Written in the frame of the diploma thesis (advisor: Prof. Gert-Martin Greuel) 'Computations of moduli spaces of semiquasihomogeneous singularities and an implementation in Singular'

```
PROCEDURES:
 HilbertSeries(I, w);             Hilbert series of the ideal I w.r.t. weight w
 HilbertWeights(I, w);            weighted degrees of the generators of I
 ImageVariety(I, F);              ideal of the image variety F(variety(I))
 ImageGroup(G, F);                ideal of G w.r.t. the induced representation
 InvariantRing(G, Gaction);       generators of the invariant ring of G
 InvariantQ(f, G, Gaction);       decide if f is invariant w.r.t. G
 LinearizeAction(G, Gaction);     linearization of the action 'Gaction' of G
 LinearActionQ(action,s,t);       decide if action is linear in var(s..nvars)
 LinearCombinationQ(base, f);     decide if f is in the linear hull of 'base'
 MinimalDecomposition(f,s,t);     minimal decomposition of f (like coef)
 NullCone(G,act);                 ideal of the nullcone of the action 'act' of G
 ReynoldsImage(RO,f);             image of f under the Reynolds operator 'RO'
 ReynoldsOperator(G, Gaction);    Reynolds operator of the group G
 SimplifyIdeal(I[,m,s]);          simplify the ideal I (try to reduce variables)
 TransferIdeal(R,name,nA);        transfer the ideal 'name' from R to basering
```

```
LIBRARY: sing.lib     Invariants of Singularities
AUTHORS: Gert-Martin Greuel,  greuel@mathematik.uni-kl.de,
         Bernd Martin,        martin@math.tu-cottbus.de

PROCEDURES:
 codim (id1, id2);    vector space dimension of of id2/id1 if finite
 deform(i);           infinitesimal deformations of ideal i
 dim_slocus(i);       dimension of singular locus of ideal i
 is_active(f,id);     is poly f an active element mod id? (id ideal/module)
 is_ci(i);            is ideal i a complete intersection?
 is_is(i);            is ideal i an isolated singularity?
 is_reg(f,id);        is poly f a regular element mod id? (id ideal/module)
 is_regs(i[,id]);     are gen's of ideal i regular sequence modulo id?
 milnor(i);           Milnor number of ideal i; (assume i is ICIS in nf)
 nf_icis(i);          generic combinations of generators; get ICIS in nf
 qhspectrum(f,w);     spectrum numbers of w-homogeneous polynomial f
 slocus(i);           ideal of singular locus of ideal i
 tangentcone(i);      tangent cone of ideal i
 Tjurina(i);          SB of Tjurina module of ideal i (assume i is ICIS)
 tjurina(i);          Tjurina number of ideal i (assume i is ICIS)
 T_1(i);              T^1-module of ideal i
 T_2((i);             T^2-module of ideal i
 T_12(i);             T^1- and T^2-module of ideal i
```

```
LIBRARY: solve.lib    Complex Solving of Polynomial Systems
AUTHOR:  Moritz Wenk,   wenk@mathematik.uni-kl.de
         Wilfred Pohl,  pohl@mathematik.uni-kl.de

PROCEDURES:
 laguerre_solve(p,[..]); find all roots of univariate polynomial p
 solve(i,[..]);          all roots of 0-dim. ideal i using triangular sets
 ures_solve(i,[..]);     find all roots of 0-dimensional ideal i with resultants
 mp_res_mat(i,[..]);     multipolynomial resultant matrix of ideal i
 interpolate(p,v,d);     interpolate poly from evaluation points p and results v
 fglm_solve(i,[..]);     find roots of 0-dim. ideal using FGLM and lex˙solve
 lex_solve(i,p,[..]);    find roots of reduced lexicographic standard basis
 triangLf_solve(l,[..]); find roots using triangular sys. (factorizing Lazard)
 triangM_solve(l,[..]);  find roots of given triangular system (Moeller)
 triangL_solve(l,[..]);  find roots using triangular system (Lazard)
 triang_solve(l,p,[..]); find roots of given triangular system
```

```
LIBRARY: spcurve.lib  Deformations and Invariants of CM-codim 2 Singularities
AUTHOR:  Anne Frühbis-Krüger,  anne@mathematik.uni-kl.de

PROCEDURES:
 isCMcod2(i);          presentation matrix of the ideal i, if i is CM
 CMtype(i);            Cohen-Macaulay type of the ideal i
 matrixT1(M,n);        1-st order deformation T1 in matrix description
 semiCMcod2(M,T1);     semiuniversal deformation of maximal minors of M
```

`discr(sem,n);`	discriminant of semiuniversal deformation
`qhmatrix(M);`	weights if M is quasihomogeneous
`relweight(N,W,a);`	relative matrix weight of N w.r.t. weights (W,a)
`posweight(M,T1,i);`	deformation of coker(M) of non-negative weight
`KSpencerKernel(M);`	kernel of the Kodaira-Spencer map

```
LIBRARY: spectrum.lib
AUTHOR:  Stefan Endraß
```
Singularity Spectrum for Nondegenerate Singularities

`PROCEDURES:`

`spectrumnd(poly[,1]);`	spectrum of a nondegenerate isolated singularity
`semicont(s1,s2[,1]);`	tests if s2 is semicontinuous for s1
`semicontqh(s1,s2);`	semicontinuity test using open and half open intervals
`spadd(s1,s2);`	sum of two spectra s1 and s2
`spmul(s,k);`	product of spectrum s with integer k

`LIBRARY: standard.lib` Procedures, which are always loaded at Start-up

`PROCEDURES:`

`stdfglm(ideal[,ord])`	standard basis of ideal via fglm [and ordering ord]
`stdhilb(ideal[,h])`	standard basis of ideal using the Hilbert function
`groebner(ideal/module)`	standard basis using a heuristically chosen method
`quot(any,any[,n])`	quotient using heuristically chosen method
`res(ideal/module,[i])`	free resolution of ideal or module
`sprintf(fmt,...)`	returns formatted string
`fprintf(link,fmt,..)`	writes formatted string to link
`printf(fmt,...)`	displays formatted string

`LIBRARY: stratify.lib` Algorithmic Stratification for Unipotent Group-Actions
`AUTHOR:  Anne Frühbis-Krüger,  anne@mathematik.uni-kl.de`

`OVERVIEW:`

This library provides an implementation of the algorithm of Greuel and Pfister introduced in the article "Geometric quotients of unipotent group actions".

`PROCEDURES:`

`prepMat(M,wr,ws,step);`	list of submatrices corresp. to given filtration
`stratify(M,wr,ws,step);`	algorithmic stratification (main procedure)

`LIBRARY: surf.lib` Procedures for Graphics with surf
```
AUTHOR:  Hans Schönemann,  hannes@mathematik.uni-kl.de,
         the programme surf is written by Stefan Endraß
```

`NOTE:`

To use this library requires the programme surf to be installed. surf is only available for Linux PCs and Sun workstations. You can download surf either from
`http://sourceforge.net/projects/surf`
or from `ftp://www.mathematik.uni-kl.de/pub/Math/Singular/utils/`.

```
PROCEDURES:
 plot(I,[...]);   plots plane curves and surfaces


LIBRARY: teachstd.lib    Procedures for Teaching Standard Bases
AUTHOR:  Gert-Martin Greuel,  greuel@mathematik.uni-kl.de

NOTE:
```

The library is intended to be used for teaching purposes only. The procedures are implemented exactly as described in the book 'A SINGULAR Introduction to Commutative Algebra' by G.-M. Greuel and G. Pfister. Sufficiently high printlevel allows to control each step.

```
PROCEDURES:
 ecart(f);              ecart of f
 tail(f);               tail of f
 sameComponent(f,g);    test for same module component of lead(f) and lead(g)
 leadmonomial(f);       leading monomial as poly (also for vectors)
 monomialLcm(m,n);      lcm of monomials m and n as poly (also for vectors)
 spoly(f[,1]);          s-polynomial of f [symmetric form]
 minEcart(T,h);         element g ∈ T of minimal ecart s.t. LM(g) divides LM(h)
 NFMora(i);             normal form of i w.r.t Mora algorithm
 prodcrit(f,g);         test for product criterion
 chaincrit(f,g,h);      test for chain criterion
 pairset(G);            pairs form G neither satisfying prodcrit nor chaincrit
 updatePairs(P,S,h);    pairset P enlarged by not useless pairs (h,f), f in S
 standard(id);          standard basis of ideal/module
 localstd(id);          local standard basis of id using Lazard's method


LIBRARY: template.lib   A Template for a Singular Library
AUTHOR:  Olaf Bachmann,  obachman@mathematik.uni-kl.de

PROCEDURES:
 mdouble(int);           returns double of int argument
 mtripple(int);          returns three times int argument
 msum([int,..,int]);     sum of int arguments

LIBRARY: toric.lib   Standard Basis of Toric Ideals
AUTHOR:  Christine Theis,  ctheis@math.uni-sb.de

PROCEDURES:
 toric_ideal(A,..);   computes the toric ideal of A
 toric_std(ideal I);  standard basis of I by a specialized Buchberger algorithm


LIBRARY: triang.lib  Decompose Zero-dimensional Ideals into Triangular Sets
AUTHOR:  Dietmar Hillebrand

PROCEDURES:
 triangL(G);          Decomposition of (G) into triangular systems (Lazard).
 triangLfak(G);       Decomp. of (G) into tri. systems plus factorization.
 triangM(G[,.]);      Decomposition of (G) into triangular systems (Möller).
```

```
 triangMH(G[,.]);      Decomp. of (G) into tri. syst. with disjoint varieties.
```

```
LIBRARY: weierstr.lib    Procedures for the Weierstraß Theorems
AUTHOR:  Gert-Martin Greuel,  greuel@mathematik.uni-kl.de

PROCEDURES:
 weierstr_div(g,f,d);  perform Weierstrass division of g by f up to gegree d
 weierstr_prep(f,d);   perform Weierstrass preparation of f up to gegree d
 lastvar_general(f);   make f general of finite order w.r.t. last variable
 general_order(f);     compute integer b s.t. f is x^n-general of order b
```

```
LIBRARY: zeroset.lib   Procedures For Roots and Factorization
AUTHOR:  Thomas Bayer,  bayert@in.tum.de

OVERVIEW:
```

Algorithms for finding the zero–set of a zero–dimensional ideal in $\mathbb{Q}(a)[x_1,..,x_n]$, Roots and Factorization of univariate polynomials over $\mathbb{Q}(a)[t]$ where a is an algebraic number. Written in the frame of the diploma thesis (advisor: Prof. Gert-Martin Greuel) 'Computations of moduli spaces of semiquasihomogeneous singularities and an implementation in Singular'. This library is meant as a preliminary extension of the functionality of Singular for univariate factorization of polynomials over simple algebraic extensions in characteristic 0. Subprocedures with postfix 'Main' require that the ring contains a variable 'a' and no parameters, and the ideal 'mpoly', where 'minpoly' from the basering is stored.

```
PROCEDURES:
 EGCD(f,g);             gcd over an algebraic extension field of Q
 Factor(f);             factorization of f over an algebraic extension field
 Quotient(f,g);         quotient q  of f w.r.t. g (in f = q·g + remainder)
 Remainder(f,g);        remainder of the division of f by g
 Roots(f);              computes all roots of f in an extension field of Q
 SQFRNorm(f);           norm of f (f must be squarefree)
 ZeroSet(I);            zero-set of the 0-dim. ideal I

AUXILIARY PROCEDURES:
 EGCDMain(f,g);         gcd over an algebraic extension field of Q
 FactorMain(f);         factorization of f over an algebraic extension field
 InvertNumberMain(c);   inverts an element of an algebraic extension field
 QuotientMain(f,g);     quotient of f w.r.t. g
 RemainderMain(f,g);    remainder of the division of f by g
 RootsMain(f);          computes all roots of f, might extend the groundfield
 SQFRNormMain(f);       norm of f (f must be squarefree)
 ContainedQ(data,f);    f in data ?
 SameQ(a,b);            a = b (list a,b)?
```

B.9 SINGULAR and Maple

In this section we give two examples how to use SINGULAR as a support for a Maple session and one example showing how to use Maple in a SINGULAR session. The first example is the trivial way, the second is based on a simplified version of a script of G. Kemper.[6]

Assume we are in a Maple session and want to compute a Gröbner basis with SINGULAR of the ideal $I = \langle x^{10} + x^9y^2, y^8 - x^2y^7\rangle$ in characteristic 0 with the degree reverse lexicographical ordering `dp`.

The first solution is to write the polynomials to the file `singular_input` (already in the SINGULAR language). This is done by the following:

```
f:=x^10+x^9*y^2;
g:=y^8-x^2*y^7;

interface(prettyprint=0);
interface(echo=0);
writeto( singular_input );
lprint(`ideal I = `);
f, g ;
lprint(`;`);
writeto(terminal);
```

The resulting file looks like:

```
ideal I =
x^10+x^9*y^2, y^8-x^2*y^7
;
```

Now we can start SINGULAR, and perform the following

```
ring R=0,(x,y),dp;
< "singular_input";
short=0;          // output in Maple format
ideal J=std(I);
write(":w maple_input",J);
```

This SINGULAR session writes the computed Gröbner basis (in Maple format) to the file `maple_input`:

```
x^2*y^7-y^8,x^9*y^2+x^10,x^12*y+x*y^11,x^13-x*y^12,y^14+x*y^12,
x*y^13+y^12
```

A more advanced solution is given by the following procedure. Here `I` is a list of polynomials, `P` is the characteristic of the ground field, and `tord` is a string specifying the SINGULAR ordering (for instance `dp`, `lp`, ...).[7]

[6] Warning: The scripts run only on Unix like operating systems.

[7] This procedure works with Maple V Release 5. In older versions of Maple, string expression were enclosed in a pair of back quotes ‘ ‘ instead of " "; moreover, the nullary operator was denoted by " instead of %. The directory `EXAMPLES/` on the enclosed CD contains two versions of the procedure – one for Maple V Release 5 and one for Maple V Release 3 (with the old syntax).

```
SINGULARlink:=proc(I,P,tord)
  local i,j,path,vars,F,p,ele;

  F:=map(expand,I);
  vars:=indets(F);
  if P>0 then p:=P else p:=0 fi;

  path:="Trans";
  if assigned(pathname) then path:=cat(pathname,path) fi;
  if system("mkdir ".path)<>0 then
    ERROR("Couldn't make the Transfer-directory")
  fi;

  # produce input for Singular ...
  writeto(cat(path,"/In"));

  # Define the ring (with term order)
  lprint(`ring R =`);
  lprint(``.p.`,`);
  lprint(`(x(1..`.(nops(vars)).`)),`);
  lprint(`(`.tord.`);`);

  # Define the ideal ...
  lprint(`ideal I =`);
  for i to nops(F) do
      if i>1 then lprint(`,`) fi;
      ele:=subs([seq(vars[j]=x(j), j=1..nops(vars))],F[i]);
      if type(ele,monomial) then lprint(ele)
      else # split into summands, otherwise line might get chopped!
          lprint(op(1,ele));
          for j from 2 to nops(ele) do
              lprint(`+`,op(j,ele))
          od
      fi
  od;
  lprint(`;`);

  lprint(`short=0;`);
  lprint(`ideal b = std(I);`);
  lprint(`write(\"`.path.`/Out\",\"[\");`);
  lprint(`write(\"`.path.`/Out\",b);`);
  lprint(`write(\"`.path.`/Out\",\"];\");`);
  lprint(`quit;`);
  writeto(terminal);

  # Call Singular ...
  system("Singular < ".path."/In > ".path."/temp");
  if %<>0 then
      system("'rm' -r ".path);
      ERROR("Something went wrong while executing Singular")
  fi;

  # Retrieve the results ...
```

```
  read(cat(path,"/Out"));
  F:=%;
  system("'rm' -r ".path);
  F:=subs([seq(x(i)=vars[i],i=1..nops(vars))],F);
  F:=map(expand,F);
  F
end:
```

Let's apply this procedure to an example:

```
f:=x^10+x^9*y^2;
g:=y^8-x^2*y^7;
J:=SINGULARlink([f,g],0,"dp"):

interface(prettyprint=0);
J;
#-> [x^2*y^7-y^8, x^10+x^9*y^2, x^12*y+x*y^11, x^13-x*y^12,
#->  y^14+x*y^12, x*y^13 +y^12]
```

Assume now that we are in a SINGULAR session and want to use Maple to factorize a polynomial. This can be done by using the following SINGULAR procedure:

```
proc maple_factorize(poly p)
{
  int saveshort=short;
  short=0;
  string in="maple-in."+string(system("pid"));
  string out="maple-out."+string(system("pid"));
  link l=in;
  write(l,"res:=factors("+string(p)+");");
  write(l,"interface(prettyprint=0);");
  write(l,"interface(echo=0);");
  write(l,"writeto(`"+out+"`);");
  write(l,"lprint(`ideal fac=`);");
  write(l,"lprint(op(1,res));");
  write(l,"for i to nops(op(2,res)) do");
  write(l,"  lprint(`,`);");
  write(l,"  lprint(op(1,op(i,op(2,res))));");
  write(l,"od;");
  write(l,"lprint(`;`);");
  write(l,"lprint(`intvec multies=`);");
  write(l,"lprint(`1`);");
  write(l,"for i to nops(op(2,res)) do");
  write(l,"  lprint(`,`);");
  write(l,"  lprint(op(2,op(i,op(2,res))));");
  write(l,"od;");
  write(l,"lprint(`;`);");
  write(l,"lprint(`list res=fac,multies;`);");
  write(l,"writeto(terminal);");
  write(l,"quit;");
  int dummy=system("sh","maple <"+in+" > dummy");
  if (dummy <> 0) { ERROR("something went wrong"); }
  string r=read(out);
```

```
  execute(r);
  return(res);
}
```

Here comes an example:

```
ring R = 0,(x,y),dp;
poly f = 5*(x-y)^2*(x+y);

maple_factorize(f);
//-> [1]:
//->    _[1]=5
//->    _[2]=x+y
//->    _[3]=-x+y
//-> [2]:
//->    1,1,2
```

B.10 Singular and Mathematica

We show by an example how to use Singular as support for a Mathematica session.[8] Assume we are in a Mathematica session and want to compute a Gröbner basis with Singular of the ideal $\langle x^{10}+x^9y^2, y^8-x^2y^7\rangle$ in characteristic 0 with the degree reverse lexicographical ordering `dp`.

```
SINGULARlink[J_List,P_Integer,tord_String] := Module[
  {i,vars,F,p,subst,varnames,SINGULARin,SINGULARout},
  F=J;
  vars=Variables[F];
  (* Substitution of Variable names *)
  varnames = Table[ ToExpression[ "x[" <> ToString[i] <> "]"],
                  {i, 1, Length[vars]} ];
  subst = Dispatch[MapThread[Rule, {vars, varnames} ]];
  F = F /. subst;
  singF = ToString[F // InputForm];
  singF = StringReplace[ singF, { "{"->"", "}"->"", ", "->",\n",
                               "]"->")", "["->"(" } ];
  If[P>0, p=P, (*Else*) p=0];
  (* Prepare Singular input string *)
  SINGULARin = "ring R =" <> ToString[p] <> ", (" <>
             "x(1.." <> ToString[Length[vars]] <> ")), (" <>
             tord <> ");\n";
  (* Define the ideal... *)
  SINGULARin = SINGULARin <> "ideal I =" <> singF <> " ;\n" <>
              "short=0;\n" <>
              "ideal b = std(I);\n" <>
              "write(\".tmp.sing.mathematica\",b );\n" <>
              "quit;\n";

  (* Call Singular ... *)
  SINGULARin // OutputForm >> "!Singular -q";
```

[8] Warning: The scripts run only on Unix like operating systems.

```
  SINGULARout=ReadList[".tmp.sing.mathematica",String];
  >>"!rm .tmp.sing.mathematica";
  SINGULARout= StringReplace[SINGULARout , {  ")"->"]",
                                              "("->"[" } ];
  SINGULARout= "{" <> SINGULARout <> "}";
  SINGULARout= ToExpression[SINGULARout];
  subst = Dispatch[MapThread[Rule, {varnames, vars} ]];
  SINGULARout=SINGULARout/. subst;
  SINGULARout
]
```

Let's apply this

```
f=x^10+x^9*y^2;
g=y^8-x^2*y^7;
J=SINGULARlink[{f,g},0,"dp"];

J // InputForm
(*-> //InputForm=
     {x^2*y^7 - y^8, x^10 + x^9*y^2, x^12*y + x*y^11,
      x^13 - x*y^12, x*y^12 + y^14, y^12 + x*y^13}
*)
```

B.11 Singular and MuPAD

Finally, we give an example how to use Singular as support for a MuPAD session.[9] Assume we are in a MuPAD session and want to compute a Gröbner basis with Singular of the ideal $\langle x^{10}+x^9y^2, y^8-x^2y^7\rangle$ in characteristic 0 with the degree reverse lexicographical ordering `dp`.

```
SINGULARlink := proc(L:DOM_LIST, P:DOM_INT, tord)
  local i, path, ipath, opath, tpath, fd, vars, F, p;

begin
  F := map(L, expand);
  vars := indets(F);

  if P > 0 then
    p := P;
  else
    p := 0;
  end_if;

  // Create the directory where "communication files" are stored
  path := "Trans";

  if singpath <> hold(singpath) then
    path := pathname(singpath, path) ;
```

[9] We should like to thank Torsten Metzner for providing the scripts. Warning: The scripts run only on Unix like operating systems.

```
    end_if;

    if system("mkdir ".path) <> 0 then
      error("Could not make the Transfer-directory")
    end_if;

    // produce input for Singular ...
    ipath := pathname(path)."In" ;
    if version() = [2, 0, 0] then
      fd := fopen(Text, ipath, Write) ;
    else // MuPAD 2.5
      fd := fopen(ipath, Write, Text) ;
    end_if ;

    // Define the ring (with term order)
    fprint(NoNL, fd, "ring R = ");
    fprint(NoNL, fd, p,", (");
    fprint(NoNL, fd, vars[i], ", " ) $i=1..nops(vars)-1;
    fprint(NoNL, fd, vars[nops(vars)], "), ");
    fprint(Unquoted, fd, "( ",tord," ); ");

    // Define the ideal ...
    fprint(Unquoted, fd, "ideal I =");
    fprint(Unquoted, fd, F[i], ", " ) $i=1..nops(F)-1;
    fprint(Unquoted, fd, F[nops(F)], ";");

    fprint(Unquoted, fd, "short=0;");
    fprint(Unquoted, fd, "ideal b = std(I);");
    opath := pathname(path)."Out" ;
    fprint(Unquoted, fd, "write(\"".opath."\",\"[\");");
    fprint(Unquoted, fd, "write(\"".opath."\",b);");
    fprint(Unquoted, fd, "write(\"".opath."\",\"];\");");
    fprint(Unquoted, fd, "quit;");
    fclose(fd);

    // Call Singular ...
    tpath := pathname(path)."temp" ;
    if system("Singular < ".ipath."> ".tpath) <> 0 then
      system("rm -r ".path);
      error("Something went wrong while executing Singular");
    end_if;

    // Retrieve the results ...
    F := read(opath, Quiet):
    system("rm -r ".path);
    F := map(F,expand);
  end_proc:
```

Now apply this procedure:

```
f:=x^10+x^9*y^2;
g:=y^8-x^2*y^7;
J:=SINGULARlink([f,g],0,"dp"):
output::tableForm(J,",");
```

```
//-> x^2*y^7 - y^8,x^10 + x^9*y^2,x*y^11 + x^12*y,x^13 - x*y^12,
//-> y^14 + x*y^12,y^12 + x*y^13
```

Assume now that we are in a Singular session and want to use MuPAD to factorize a polynomial. This can be done by using the following Singuar procedure:

```
proc mupad_factorize(poly p)
{
  int saveshort = short;
  short = 0;
  string in  = "mupad-in."+string(system("pid"));
  string out = "mupad-out."+string(system("pid"));
  link l = in;
  write(l, "res:=factor("+string(p)+");");
  write(l, "if version() = [2, 0, 0] then ");
  write(l, "fd := fopen(Text, \""+out+"\", Write) ;");
  write(l, "else");
  write(l, "fd := fopen(\""+out+"\", Write, Text) ;");
  write(l, "end_if;");

  write(l,"fprint(Unquoted, fd, \"ideal fac=\");");
  write(l,"fprint(Unquoted, fd, op(res,1));");
  write(l,"for i from 2 to nops(res) step 2 do");
  write(l,"  fprint(Unquoted, fd, \",\", op(res,i));");
  write(l,"end_for;");
  write(l,"fprint(Unquoted, fd, \";\");");

  write(l,"fprint(Unquoted, fd, \"intvec multies=\");");
  write(l,"fprint(Unquoted, fd, \"1\");");
  write(l,"for i from 3 to nops(res) step 2 do");
  write(l,"  fprint(Unquoted, fd, \",\", op(res,i));");
  write(l,"end_for;");
  write(l,"fprint(Unquoted, fd, \";\");");

  write(l,"fprint(Unquoted, fd, \"list res=fac,multies;\");");
  write(l,"fclose(fd);");
  write(l,"quit;");
  int dummy = system("sh","mupad <"+in+" > dummy");
  if (dummy <> 0) { ERROR("something went wrong"); }
  string r=read(out);
  execute(r);
  return(res);
}
```

We apply this procedure to an example:

```
ring R = 0,(x,y),dp;
poly f = 5*(x-y)^2*(x+y);

mupad_factorize(f);
//-> [1]:
//->    _[1]=5
```

```
//->      _[2]=x+y
//->      _[3]=-x+y
//->[2]:
//->     1,1,2
```