Singular

A Computer Algebra System for Polynomial Computations

Manual

Version 1.2

Singular is created and its development is directed and coordinated by
G.-M. Greuel, G. Pfister, and H. Schoenemann

with contributions by
O. Bachmann, W. Decker, C. Gorzel, H. Grassmann, A. Heydtmann, K. Krueger, M. Lamm,
B. Martin, M. Messollen, W. Neumann, T. Nuessler, W. Pohl, T. Siebert, R. Stobbe, T. Wichmann

Fachbereich Mathematik
und
Zentrum fuer Computeralgebra
Universitaet Kaiserslautern
D-67653 Kaiserslautern

1 Preface

SINGULAR version 1.2
University of Kaiserslautern
Department of Mathematics and Centre for Computer Algebra
Authors: G.-M. Greuel, G. Pfister, H. Schoenemann

NOTICE

If you use Singular or parts thereof in a project and/or publish results that were partly obtained using SINGULAR, we request that you cite SINGULAR (see `http://www.mathematik.uni-kl.de/~zca/Singular/how_to_cite.html` on information on how to cite Singular) and inform us about it.

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

1.1 Availability

The latest information about SINGULAR is always available from `http://www.mathematik.uni-kl.de/~zca/Singular`. The program SINGULAR and the above mentioned parts are available via anonymous ftp through the following addresses:

GMP © Free Software Foundation: `libgmp-2.0.2.tar.gz`
`ftp://ftp.gnu.ai.mit.edu` or its mirrors

readline © Free Software Foundation: `libreadline-2.0.tar.gz`
`ftp://ftp.gnu.ai.mit.edu` or its mirrors

MP © Gray/Kajler/Wang, Kent State University: `MP-1.1.2.tar.gz`,
`http://SymbolicNet.mcs.kent.edu/areas/protocols/mp.html`

Factory © Greuel/Stobbe, University of Kaiserslautern: `factory-1.3b.tar.gz`
Contact `factory@mathematik.uni-kl.de` for details on Factory.

libfac © Messollen, University of Saarbrücken: `libfac-0.3.0.tar.gz`
Contact `michael@math.uni-sb.de` for details on libfac.

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

1.2 Acknowledgements

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

Contributions to the kernel of SINGULAR have been made by Olaf Bachmann, Hubert Grassmann, Kai Krüger, Wolfgang Neumann, Thomas Nüßler, Wilfred Pohl, Jens Schmidt, Thomas Siebert, Rüdiger Stobbe, and Tim Wichmann.

SINGULAR libraries have furthermore been contributed by Wolfram Decker, Christian Gorzel, Agnes Eileen Heydtmann, Ulrich Klein, Martin Lamm, and Bernd Martin.

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

2 Introduction

2.1 Background

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

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

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

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

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

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

Continuous extensions (like polynomial factorization, gcd computations, links) and refinements led in 1997 to the release of SINGULAR version 1.0.

The highlights of the new SINGULAR version 1.2. include: much faster standard and Groebner bases computations based on Hilbert series and on improved implementations of the algorithms and the addition of libraries for primary decomposition, ring normalization, etc.

2.2 How to use this manual

For the impatient user

In Section 2.3 [Getting started], page 5, some simple examples are explained in a step-by-step manner to introduce into SINGULAR.

Appendix A [Examples], page 196 should come next for real learning-by-doing or to quickly solve some given mathematical problems without dwelling to deeply into SINGULAR. This chapter contains a lot of real-life examples and detailed instructions and explanations on how to solve mathematical problems using SINGULAR.

For the systematic user

In Chapter 3 [General concepts], page 16, all basic concepts which are important to use and understand SINGULAR are developed. But even for users preferring the systematic approach it will be helpful to have a look at the examples in Section 2.3 [Getting started], page 5, every now and then. The topics in the chapter are organized more or less in the order the novice user has to deal with them.

- In Section 3.1 [Interactive use], page 16, and its subsections there are some words on entering and exiting SINGULAR, followed by a number of other aspects concerning the interactive user-interface.
- To do anything more than trivial integer computations, one needs to define a basering in SINGULAR. This is explained in detail in Section 3.2 [Rings and orderings], page 20.
- An overview of the algorithms implemented in the kernel of SINGULAR is given in Section 3.3 [Implemented algorithms], page 25.
- In Section 3.4 [The SINGULAR language], page 29, language specific concepts are introduced such as the notions of names and objects, data types and conversion between them, etc.
- In Section 3.5 [Input and output], page 36, SINGULAR's mechanisms to store and retrieve data are discussed.
- The more complex concepts of procedures and libraries as well as tools to debug them are considered in the following sections: Section 3.6 [Procedures], page 40, Section 3.7 [Libraries], page 44, and Section 3.8 [Debugging tools], page 46.

Chapter 4 [Data types], page 49, is a complete treatment for SINGULAR's data types where each section corresponds to one data type, alphabetically sorted. For each data type, its purpose is explained, the syntax of its declaration is given, and related operations and functions are listed. Examples illustrate its usage.

Chapter 5 [Functions and system variables], page 102, is an alphabetically ordered reference list of all of SINGULAR's functions, control structures, and system variables. Each entry includes a description of the syntax and semantics of the item being explained as well as one or more examples on how to use it.

Miscellaneous

Chapter 6 [Tricks and pitfalls], page 190, is a loose collection of limitations and features which may be unexpected by those who expect to be the SINGULAR language an exact copy of the C programming language or of some Computer Algebra system languages. But some mathematical tips are collected there ase well.

Appendix C [Mathematical background], page 246 introduces some of the mathematical notions and definitions used throughout this manual. For example, if in doubt what exactly SINGULAR means by a "negative degree reverse lexicographical ordering" one should refer to this chapter.

Appendix D [SINGULAR libraries], page 250, and Appendix E [Library function index], page 259 lists the libraries which come with SINGULAR and the functions contained in them, resp.

Typographical conventions

Throughout this manual, the following typographical conventions are adopted:

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

  The basering can be set using the command `setring`.
- the arrow ↦ denotes SINGULAR output:

  ```
  poly p=x+y+z;
  p*p;
  ↦ x2+2xy+y2+2xz+2yz+z2
  ```
- square brackets are used to denote parts of syntax descriptions which are optional:

  [optional_text] required_text
- keys are denoted using typewriter, for example:

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

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

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

2.3 Getting started

SINGULAR is a special purpose system for polynomial computations. Hence, the most powerful computations in SINGULAR require the prior definition of a ring. Most important rings are polynomial rings over a field, localizations hereof, 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.

2.3.1 First steps

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

```
37+5;
↦ 42
```

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

```
int k = 2;
```

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

```
k == 2;
↦ 1
k != 2;
↦ 0
```

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

```
k;
↦ 2
```

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

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

The last displayed (!) result is always available with the special symbol `_` .

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

Text starting with `//` denotes a comment and is ignored in calculations, as seen in the previous example. Furthermore SINGULAR maintains a history of the previous lines of input, which may be accessed by `CTRL-P` (previous) and `CTRL-N` (next) or the arrows on the keyboard. Note, that the history is not available on Macintosh systems.

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

```
help intmat;
```

This shows the text of Section 4.4 [intmat], page 60, in the printed manual.

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

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

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

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

To calculate the trace of this matrix, we use a `for` loop. The curly brackets (`{` and `}`) denote the beginning resp. end of a block. If you define a variable without giving an initial value, as the variable `tr` in the example below, SINGULAR assigns a default value for the specific type. In this

case, the default value for integers is 0. Note, that the integer variable j has already been defined above.

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

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

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

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

2.3.2 Rings and standard bases

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

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

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

Other ring declarations:

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

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

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

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

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

```
ring r5;
r5;
↦ //   characteristic : 32003
↦ //   number of vars : 3
↦ //        block   1 : ordering dp
↦ //                  : names    x y z
↦ //        block   2 : ordering C
```

Defining a ring makes this ring the current active basering, so each ring definition above switches to a new basering. The concept of rings in SINGULAR is discussed in detail in Section 3.2 [Rings and orderings], page 20.

The basering now is `r5`. Since we want to calcualate in the ring `r`, which we defined first, we have to switch back to it. This can be done using the function `setring`:

```
setring r;
```

Once a ring is active, we can define polynomials. A monomial, say x^3 may be entered in two ways: either using the power operator `^`, saying `x^3`, or in short-hand notation without operator, saying `x3`. Note, that the short-hand notation is forbidden if the name of the ring variable consists of more than one character. Note, that SINGULAR always expands brackets and automatically sorts the terms with respect to the monomial ordering of the basering.

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

The command `size` determines in general the number of "single entries" in an object. In particular, for polynomials, `size` determines the number of monomials.

```
size(f);
↦ 5
```

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

```
poly g =  f^2 *(2x-y);
ideal I = f,g;
ideal J= subst(I,var(1),1);
```

```
J = subst(J,var(2),2);
J = subst(J,var(3),0);
J;
↦ J[1]=5
↦ J[2]=0
```

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

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

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

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

The function `groebner`, like many other functions in SINGULAR, prints a protocol during calculation, if desired. The command `option(prot);` enables protocoling whereas `option(noprot);` turns it off. Section 5.1.78 [option], page 150, explains the meaning of the different symbols printed during calculation.

The command `kbase` calculates a basis of the polynomial ring modulo an ideal, if the quotient ring is finite dimensional. As an example we calculate the Milnor number of a hypersurface singularity in the global and local case. This is the vector space dimension of the polynomial ring modulo the Jacobian ideal in the global case resp. of the power series ring modulo the Jacobian ideal in the local case. See Section A.3 [Critical points], page 199, for a detailed explanation.

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

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

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

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

```
J = groebner(J);
ideal K = kbase(J);
K;
↦ K[1]=y4
↦ K[2]=xy3
↦ K[3]=y3
```

```
↦ K[4]=xy2
↦ K[5]=y2
↦ K[6]=x2y
↦ K[7]=xy
↦ K[8]=y
↦ K[9]=x3
↦ K[10]=x2
↦ K[11]=x
↦ K[12]=1
```

Then

```
size(K);
↦ 12
```

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

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

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

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

2.3.3 Procedures and libraries

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

Define the following procedure called `Milnor`:

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

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

Then call the procedure:

```
Milnor(f);
↦ 12
```

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

The distribution of SINGULAR contains several libraries, each of which is a collection of useful procedures based on the kernel commands, which extend the functionality of SINGULAR. The command `help "all.lib";` lists all libraries together with a one-line explanation.

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

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

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

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

```
milnor(f);
↦ 12
```

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

```
help milnor;
```

as well as some examples, which are executed by

```
example milnor;
```

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

```
help sing.lib;
```

The output of the help commands is omitted here.

2.3.4 Change of rings

To calculate the local Milnor number we have to do the calculation with the same commands in a ring with local ordering. Define the localization of the polynomial ring at the origin (see Appendix B [Polynomial data], page 239 and Appendix C [Mathematical background], page 246).

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

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

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

Instead of `fetch` we can use the function `imap` which is more general but less efficient. The most general way to fetch data from one ring to another is to use maps, this will be explained in Section 4.8 [map], page 78.

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

```
Milnor(f);
↦ 4
```

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

The command `jacob` applied twice gives the Hessian of `f`, a 3×3-matrix.

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

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

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

We may calculate the determinant and minors of different size.

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

```
↦ _[5]=6x+6xy2-2y3
```

The variety defined by the 1×1-minors is empty. The algorithm of the standard basis computations may be affected by the command `option`. For instance, a reduced standard basis is obtained in the following way

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

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

2.3.5 Modules and their annihilator

Now we shall give now three more advanced examples.

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

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

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

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

However the submodule MD may also be considered as the module of relations of the factor module r^3/MD. In this way, SINGULAR can treat arbitrary finitely generated modules over the basering (see Section B.1 [Representation of mathematical objects], page 239).

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

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

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

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

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

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

The result is too big to be shown here.

2.3.6 Resolution

There are several commands in SINGULAR for computing free resolutions. The command `mres(... ,n)`, for example, calculates a minimal free resolution with the standard basis method up to the length n, where $n = 0$ corresponds to the full resolution.

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

```
ring rr;      // the default ring in char 32003
rr;
↦ //   characteristic : 32003
↦ //   number of vars : 3
↦ //        block   1 : ordering dp
↦ //                  : names    x y z
↦ //        block   2 : ordering C
ideal I = x4+x3y+x2yz,x2y2+xy2z+y2z2,x2z2+2xz3,2x2z2+xyz2;
option(prot);
resolution rs = mres(I,0);
↦ v4(3)s-s.s.s5(3)s6-s.s-s7(2)s-s-s
↦ product criterion:0 chain criterion:8
↦ (6)(5).(4).(3)..(2)..(1).[1]
↦ (7)(6)(5)(4).(3)(2).(1)[2]
↦  // .... rest of protocol omitted here
```

Disable this protocol with

```
option(noprot);
```

When we enter the name of the calculated resolution, we get a pictorial description of the resolution where the exponents denote the rank of the free modules.

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

A single module in this resolution is obtained as usual with the brackets `[` and `]`.

```
print(rs[3]);
↦ y2z,  y3,
↦ -y2,  -y2,
↦ x+y-z,x+y-z,
↦ -2z,  x-y+4z,
↦ -z,   -y+3z
```

3 General concepts

3.1 Interactive use

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

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

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

3.1.1 How to enter and exit

To start SINGULAR, enter `Singular` at the system prompt. The SINGULAR banner appears which, among others, reports the version and the compilation date.

To exit SINGULAR type `quit;`, `exit;` or `$`.

SINGULAR may also be started with command line options and with filenames as arguments. More generally, the synopsis of SINGULAR is

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

See Section 3.1.6 [Command line options], page 18, Section 3.1.7 [Startup sequence], page 20.

3.1.2 The SINGULAR prompt

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

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

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

3.1.3 The online help system

The online help system is invoked with the `help` command. `?` may be used as a synonym for `help`. Simply typing `help;` displays the "top" of the help system which offers a short table of content. Typing `help` topic`;` shows the available documentation on topic. Here, topic may be

either a function name or, more generally, the name of any section of the printed manual. See Section 5.1.38 [help], page 125, for more information.

On Unix-like operating systems, the online help system is based on the `info` program from the Gnu `texinfo` package. See section "Getting started" in *The Info Manual*, for more information. On the other systems the online help system is not interactively as described in the remainder of this section. Instead, after displaying the information SINGULAR immediately returns to the input prompt.

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

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

The most important commands are:

`q`	leaves the online help system
`n`	goes to the next node
`p`	goes to the previous node
`u`	goes to the upper node
`m`	picks a menu item specified by name
`f`	follows a cross reference
`l`	goes to the previously visited node
`b`	goes to the beginning of the current node
`e`	goes to the end of the current node
`SPACE`	scrolls forward a page
`DEL`	scrolls backward a page
`h`	invokes info tutorial (use `l` to return to the manual or `CTRL-X 0` to remove extra window)
`CTRL-H`	shows a short overview on the online help system (use `l` to return to the manual or `CTRL-X 0` to remove extra window)
`s`	searches through the manual for a specified string, and select the node in which the next occurrence is found
`1`, ..., `9`	picks i-th subtopic from a menu

3.1.4 Interrupting SINGULAR

On Unix-like operating systems and on Windows NT, typing `CTRL-C` interrupts SINGULAR. SINGULAR prints the current command and the current line and prompts for further action. The following choices are available:

`a`	returns to the top level after finishing the current command
`c`	continues
`q`	quits SINGULAR

3.1.5 Editing input

This section describes only a subset of the key bindings of SINGULAR binaries built with the GNU Readline library. See section "Command Line Editing" in *The GNU Readline Library Manual*, for more information.

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

`TAB`	provides command line completion for function names and file names
`CTRL-B`	moves cursor left
`CTRL-F`	moves cursor right
`CTRL-A`	moves cursor to beginning of line
`CTRL-E`	moves cursor to end of line
`CTRL-D`	deletes character under cursor Warning: on an empty line, `CTRL-D` is interpreted as the `EOF` character which immediately terminates SINGULAR.

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

`CTRL-K`	kills from cursor to end of line
`CTRL-U`	kills from cursor to beginning of line
`CTRL-N`	saves current line on history and get next line
`CTRL-P`	saves current line on history and get previous line
`RETURN`	saves the current line to the history and sends it to the SINGULAR parser for interpretation

On Unix-like operating systems, SINGULAR maintains a history of the last lines of input. If the environment variable `SINGULARHIST` is set and has a name of a file as value, then the input history is stored across sessions using this file. Otherwise, i.e., if the environment variable `SINGULARHIST` is not set, then the history of the last inputs is only available for the commands of the current session.

3.1.6 Command line options

The synopsis of SINGULAR is

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

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

`-e, --echo[=VAL]`
: Set value of variable `echo` to `VAL` (integer in the range 0, ..., 9). Without an argument, `echo` is set to 1, which echoes all input coming from a file. By default, the value of `echo` is 0. See Section 5.3.2 [echo], page 185.

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

`--no-rc`
: Do not execute the `.singularrc` file on start-up. By default, this file is executed on start-up. See Section 3.1.7 [Startup sequence], page 20.

`--no-stdlib`
: Do not load the library `standard.lib` on start-up. By default, this library is loaded on start-up. See Section 3.1.7 [Startup sequence], page 20.

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

`--no-out`
: Suppress display of all ouput.

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

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

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

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

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

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

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

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

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

The last three options are of interest for the use with MP links:

`-b`, `--batch`
: Run in MP batch mode. Opens a TCP/IP connection with host specified by `--MPhost` at the port specified by `--MPport`. Input is read from and output is written to this connection in the MP format. See Section 4.6.5.2 [MPtcp links], page 70.

`--MPport=PORT`
: Use `PORT` as default port number for MP connections (whenever not further specified). This option is mandatory when the `--batch` option is given. See Section 4.6.5.2 [MPtcp links], page 70.

`--MPhost=HOST`
: Use `HOST` as default host for MP connections (whenever not further specified). This option is mandatory when the `--batch` option is given. See Section 4.6.5.2 [MPtcp links], page 70.

The value of options given to SINGULAR (resp. their default values, if an option was not given), can be checked with the command `system(" long_option_name_string ")`. See Section 5.1.110 [system], page 172.

```
system("--quiet");    // if ``quiet'' 1, otherwise 0
↦ 1
system("--min-time"); // minimal reported time
↦ 0.5
```

3.1.7 Startup sequence

On start-up, SINGULAR

1. loads the library `standard.lib` (provided the `--no-stdlib` option was not given),
2. searches the current directory and then the home directory of the user for a file named `.singularrc` and executes it, if found (provided the `--no-rc` option was not given),
3. executes the string specified with the `--execute` command line option,
4. executes the files `file1`, `file2` ... (given on the command line) in that order.

See Section 5.1.55 [LIB], page 136 for the directories on where SINGULAR searches for its library files.

3.2 Rings and orderings

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

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

Except for quotient rings, all of these rings are realized by choosing a coefficient field, ring variables, and an appropriate global or local monomial ordering on the ring variables. See Section 3.2.3 [Term orderings], page 24, Appendix C [Mathematical background], page 246.

The coefficient field of the rings may be

1. the field of rational numbers Q,
2. finite fields Z/p, p a prime ≤ 32003,
3. finite fields $\mathrm{GF}(p^n)$ with p^n elements, p a prime, $p^n \leq 2^{15}$,
4. transcendental extension of Q or Z/p,
5. simple algebraic extension of Q or Z/p,
6. the field of real numbers represented by simple precision floating point numbers.

Throughout this manual, the current active ring in SINGULAR is called basering. The reserved name `basering` in SINGULAR is an alias for the current active ring. The basering can be set by declaring a new ring as described in the following subsections or with the commands `setring` and `keepring`. See Section 5.2.7 [keepring], page 182; Section 5.1.100 [setring], page 164.

Objects of ring dependent types are local to a ring. To access them after a change of the basering they have to be mapped using `map` or by the functions `imap` or `fetch`. See Section 3.4.4 [Objects], page 34; Section 5.1.28 [fetch], page 119; Section 5.1.41 [imap], page 127; Section 4.8 [map], page 78.

All changes of the basering in a procedure are local to this procedure unless a `keepring` command is used as the last statement of the procedure. See Section 3.6 [Procedures], page 40, Section 5.2.7 [keepring], page 182.

3.2.1 Examples of ring declarations

The exact syntax of a ring declaration is given in the next two subsections; this subsection lists some examples first.

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

```
ring r;
ring r = 32003,(x,y,z),dp;
```

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

```
ring r = 0,(a,b,c,d),lp;
```

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

```
ring r = 7,(x,y,z),ds;
ring r = 10,(x,y,z),ds;
```

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

```
ring r = 7,(x(1..6)),(lp(3),dp);
```

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

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

- the ring $Q[x,y,z]$ with weighted reverse lexicographical ordering. The variables x, y, and z have the weights 2, 1, and 3, resp. Vectors are ordered by components first, then by monomials. Vector components are ordered in descending order. For ascending order, component ordering `C` would have been specified:

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

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

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

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

```
ring r = (7,a),(x,y,z),dp; minpoly = a^2+a+3;
ring r = (7^2,a),(x,y,z),dp;
```

- the ring $R'[x,y,z]$, where R' denotes the field of real numbers represented by simple precision floating point numbers:

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

- the quotient ring $Z/7[x,y,z]$ modulo the square of the maximal ideal (x,y,z):

```
ring R;
qring r = std(maxideal(2));
```

3.2.2 General syntax of a ring declaration

Rings

Syntax: `ring` name `=` coefficient_field`,` `(` names_of_ring_variables `),` `(` ordering `);`

Default: `32003,(x,y,z),(dp,C);`

Purpose: declares a ring and sets it as the actual basering.

The coefficient_field is given by one of the following:

1. a non-negative int_expression less or equal 32003.
   The int_expression should either be 0, specifying the field of rational numbers Q, or a prime number p, specifying the finite field with p elements. If it is not a prime number, int_expression is converted to the next lower prime number.
2. an expression_list of an int_expression and one or more names.
   The int_expression specifies the characteristic of the coefficient field as described above. The names are used as parameters in transcendental or algebraic extensions of the coefficient field. Algebraic extensions are implemented for one parameter only. In this case, a minimal polynomial has to be defined by assignment to `minpoly`. See Section 5.3.3 [minpoly], page 185.

3. an expression_list of an int_expression and a name.
   The int_expression has to be a prime number p to the power of a positive integer n. This defines the Galois field GF(p^n) with p^n elements, where p^n has to be smaller or equal 2^{15}. The given name refers to a primitive element of GF(p^n) generating the multiplicative group. Due to a different internal representation, the arithmetic operations in these coefficient fields are faster than arithmetic operations in algebraic extensions as described above.
4. the name `real`.
   This specifies the field of real numbers represented as machine floating point numbers. Note that computations over this field are not exact.

The names_of_ring_variables is a list of names or indexed names.

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

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

For the definition of the orderings, see Section 3.2.3 [Term orderings], page 24, Appendix C [Mathematical background], page 246.

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

Quotient rings

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

Default: none

Purpose: declares a quotient ring as the basering modulo ideal_expression. Sets it as current basering.

ideal_expression has to be represented by a standard basis.

The most convenient way to map objects from a ring to its quotient ring and vice versa is to use the `fetch` function (see Section 5.1.28 [fetch], page 119).

SINGULAR computes in a quotient rings as long as possible with the given representative of a polynomial, say, `f`. I.e., it usually does not reduce `f` w.r.t. the quotient ideal. This is only done when necessary during standard bases computations or by an explicate reduction using the command `reduce(f, std(0))` (see Section 5.1.94 [reduce], page 161).

Example:

```
ring r=32003,(x,y),dp;
poly f=x3+yx2+3y+4;
qring q=std(maxideal(2));
```

```
basering;
↦ //   characteristic : 32003
↦ //   number of vars : 2
↦ //        block   1 : ordering dp
↦ //                  : names    x y
↦ //        block   2 : ordering C
↦ // quotient ring from ideal
↦ _[1]=y2
↦ _[2]=xy
↦ _[3]=x2
poly g=fetch(r, f);
g;
↦ x3+x2y+3y+4
reduce(g,std(0));
↦ 3y+4
```

3.2.3 Term orderings

In a ring declaration, SINGULAR offers the following orderings:

1. Global orderings

   `lp` lexicographical ordering

   `dp` degree reverse lexicographical ordering

   `Dp` degree lexicographical ordering

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

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

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

2. Local orderings

   `ls` negative lexicographical ordering

   `ds` negative degree reverse lexicographical ordering

   `Ds` negative degree lexicographical ordering

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

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

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

3. Matrix orderings

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

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

4. Extra weight vector

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

5. Product ordering

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

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

6. Module orderings

   `(` ordering`, ..., C )`

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

   `( C,` ordering`, ... )`

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

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

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

For a mathematical description of these orderings, see Appendix B [Polynomial data], page 239.

3.3 Implemented algorithms

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

Nonetheless, there are a lot of other important algorithms:

- Algorithms to compute the standard operations on ideals and modules: intersection, ideal quotient, elimination, etc.
- Different Szyzygy algorithms and algorithms to compute free resolutions of modules.

- Combinatorial algorithms to compute dimensions, Hilbert series, multiplicities, etc.
- Algorithms for univariate and multivariate polynomial factorization, resultant and gcd computations.

Commands to compute standard bases

`facstd` Section 5.1.26 [facstd], page 118
computes a list of Groebner bases via the Factorizing Groebner Basis Algorithm, i.e. has the same radical as the original ideal. It need not be a Groebner basis of the given ideal.
The intersection of the zero sets is the zero set of the given ideal.

`fglm` Section 5.1.29 [fglm], page 121
computes a Groebner basis provided that a reduced Groebner basis w.r.t. another ordering is given.
Implements the so-called FGLM (Faugere, Gianni, Lazard, Mora) algorithm. The given ideal must be zero-dimensional.

`groebner` Section 5.1.37 [groebner], page 124
computes a standard resp. Groebner bases using a heuristically chosen method.
This is the prefered method to compute a standard resp. Groebner bases.

`mstd` Section 5.1.68 [mstd], page 145
computes a standard basis and a minimal set of generators.

`std` Section 5.1.106 [std], page 169
computes a standard resp. Groebner basis.

`stdfglm` Section 5.1.107 [stdfglm], page 170
computes a Groebner basis in a ring with a "difficult" ordering (e.g. lexicographical) via `std` w.r.t. a "simple" ordering and `fglm`.
The given ideal must be zero-dimensional.

`stdhilb` Section 5.1.108 [stdhilb], page 171
computes a Groebner basis in a ring with a "difficult" ordering (e.g. lexicographical) via `std` w.r.t. a "simple" ordering and a `std` computation guided by the Hilbert series.

Further processing of standard bases

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

`degree` Section 5.1.15 [degree], page 112
computes the (Krull) dimension, codimension and the multiplicity.
The result is only displayed on the screen.

`dim` Section 5.1.19 [dim], page 114
computes the dimension of the ideal resp. module.

`hilb` Section 5.1.39 [hilb], page 125
computes the first and resp. or second Hilbert series of an ideal resp. module.

`kbase` Section 5.1.48 [kbase], page 132
computes a vector space basis (consisting of monomials) of the quotient of a ring by an ideal resp. of a free module by a submodule.

The ideal resp. module has to be finite dimensional and has to be represented by a standard basis w.r.t. the ring ordering.

`mult` Section 5.1.69 [mult], page 145
computes the degree of the monomial ideal resp. module generated by the leading monomials of the input.

`reduce` Section 5.1.94 [reduce], page 161
reduces a polynomial, vector, ideal or module to its normal form with respect to an ideal or module represented by a standard basis.

`vdim` Section 5.1.118 [vdim], page 176
computes the vector space dimension of a ring (resp. free module) modulo an ideal (resp. module).

Commands to compute resolutions

`minres` Section 5.1.64 [minres], page 142
minimizes a free resolution of an ideal or module.

`lres` Section 5.1.59 [lres], page 139
computes a free resolution of an ideal or module with LaScala's method.

The input needs to be homogeneous.

`mres` Section 5.1.67 [mres], page 144
computes a minimal free resolution of an ideal or module with the Szyzygy method.

`res` Section 5.1.96 [res], page 162
computes a free resolution of an ideal or module using a heuristically chosen method.

This is the prefered method to compute free resolutions of ideals or modules.

`sres` Section 5.1.104 [sres], page 167
computes a free resolution of an ideal or module with Schreyer's method.

The input has to be a standard basis.

`syz` Section 5.1.111 [syz], page 173
computes the first Szyzygy (i.e., the module of relations of the given generators).

Further processing of resolutions

`betti` Section 5.1.3 [betti], page 104
computes the graded Betti numbers of a module from a free resolution.

`minres` Section 5.1.64 [minres], page 142
minimizes a free resolution of an ideal or module.

Processing of polynomials

`char_series`
Section 5.1.5 [char_series], page 106
computes characteristic sets of polynomial ideals.

`extgcd` Section 5.1.25 [extgcd], page 117
computes the extended gcd of two polynomials.

Implemented as extended Euclidean Algorithm. Applicable for univariate polynomials only.

`factorize`
Section 5.1.27 [factorize], page 119
computes factorization of univariate and multivariate polynomials into irreducible factors.

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

Note that there is no factorization of polynomials over algebraic extensions of Q.

`gcd` Section 5.1.34 [gcd], page 123
computes univariate and multivariate polynomial greatest common divisors.

For prime characteristic, a subresultant gcd is used. In characteristic 0, a modular algorithm is used for the univariate case. For the multivariate case, the EZGCD is used.

Note that there is no gcd calculation of polynomials over algebraic extensions of Q.

`resultant`
Section 5.1.98 [resultant], page 163
computes the resultant of two univariate polynomials using the subresultant algorithm.

Multivariate polynomials are considered univariate polynomials in a main variable to be specified by the user.

Matrix computations

`bareiss` Section 5.1.2 [bareiss], page 103
implements Gauss-Bareiss method for elimination (matrix triangularization) in arbitrary integral domains.

`det` Section 5.1.17 [det], page 113
computes the determinant of a square matrix.

For matrices with integer entries a modular algorithm is used. For other domains, elementary algorithms are used.

`minor` Section 5.1.63 [minor], page 141
computes all minors (=subdeterminants) of a given size for a matrix.

Controlling computations

`option` Section 5.1.78 [option], page 150
allows to set options for manipulating the behaviour of computations (such as reduction strategies) and to show protocol information indicating the progress of a computation.

3.4 The SINGULAR language

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

In the following, the basic language concepts such as commands, expressions, names, objects, etc., are discussed. See Section 3.6 [Procedures], page 40, and Section 3.7 [Libraries], page 44, for the concepts of procedures and libraries.

In many aspects, the SINGULAR language is similar to the C programming language. For a description of some of the subtle differences, see Section 6.2 [Major differences to the C programming language], page 190.

Elements of the language

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

- An "expression" is a sequence of operators, functions, and operands that specifies a computation. An expressions always results in a value of a specific type. See Chapter 4 [Data types], page 49, and its subsections (e.g., Section 4.12.2 [poly expressions], page 89), for information on how to build expressions.
- A "command" is either a declaration, an assignment, a call to a function without return value, or a print command. For detailed information, see Section 3.4.1 [General command syntax], page 29.
- "Control structures" determine the execution sequence of commands. SINGULAR provides control structures for conditional execution (`if ... else`) and iteration (`for` and `while`). Commands may be grouped in pairs of `{ }` (curly brackets) to form blocks. See Section 5.2 [Control structures], page 179, for more information.

Other notational conventions

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

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

3.4.1 General command syntax

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

Declaration

1. type name = expression ;
   declares a variable with the given name of the given type and assigns the expression as initial value to it. Expression is an expression of the specified type or one that can be converted to that type. See Section 3.4.5 [Type conversion and casting], page 34.
2. type name_list = expression_list ;
   declares variables with the given names and assigns successively each expression of expression_list to the corresponding name of name_list. Both lists must be of the same length. Each expression in expression_list is an expression of the specified type or one that can be converted to that type. See Section 3.4.5 [Type conversion and casting], page 34.
3. type name ;
   declares a variable with the given name of the given type and assigns the default value of the specific type to it.

See Section 3.4.3 [Names], page 32, for more information on declarations. See Chapter 4 [Data types], page 49, for a description of all data types known to SINGULAR.

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

Assignment

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

There must be a type conversion of the type of expression to the type of name. See Section 3.4.5 [Type conversion and casting], page 34.

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

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

Function without return value

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

The function may have output (not to be confused with a return value of type string). See Section 5.1 [Functions], page 102. Functions without a return value are specified there to have a return type 'none'.

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

```
ring r;
ideal i=x2+y2,x;
i=std(i);
degree(i);        // degree has no return value but prints output
↦ // codimension = 2
↦ // dimension   = 1
↦ // degree      = 2
```

Print command

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

Use the function `print` (or the procedure show from inout.lib) to get a pretty output of various data types, e.g., matrix or intmat. See Section 5.1.87 [print], page 156.

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

3.4.2 Special characters

The following characters and operators have special meaning:

`=`	assignment
`(, )`	in expressions, for indexed names and for argument lists
`[, ]`	access operator for strings, integer vectors, ideals, matrices, polynomials, resolutions, and lists. Used to build vectors of polynomials. Example: `s[3]`, `m[1,3]`, `i[1..3]`, `[f,g+x,0,0,1]`.
`+`	addition operator
`-`	subtraction operator
`*`	multiplication operator
`/`	division operator. See Section 6.3 [Miscellaneous oddities], page 193, for the difference between the division operators `/` and `div`.
`%`	modulo operator

`^` or `**`	exponentiation operator
`==`	comparison operator equal
`!=` or `<>`	comparison operator not equal
`>=`	comparison operator bigger or equal
`>`	comparison operator bigger
`<=`	comparison operator smaller or equal
`<`	comparison operator smaller. Also used for file input. See Section 5.1.30 [filecmd], page 121.
`!`	boolean operator not
`&&`	boolean operator and
`||`	boolean operator or
`"`	delimiter for string constants
`` ` ``	delimiter for name substitution
`?`	synonym for `help`
`//`	comment delimiter. Comment extends to end of line.
`;`	statement separator
`,`	separator for expression lists and function arguments
`\`	escape character for `"` and `\` within strings
`..`	interval specifier returning intvec. E.g., `1..3` which is equivalent to the intvec `1, 2, 3`.
`_`	value of expression last displayed
`~`	breakpoint in procedures
`#`	list of parameters in procedures without explicit parameter list
`$`	terminates SINGULAR

3.4.3 Names

SINGULAR is a strongly typed language. This means that all names (= identifiers) have to be declared prior to their use. For the general syntax of a declaration, see the description of declaration commands (see Section 3.4.1 [General command syntax], page 29).

See Chapter 4 [Data types], page 49, for a description of SINGULAR's data types. See Section 5.1.115 [typeof], page 175, for a short overview of possible types. To get information on a name and the object named by it, the `type` command may be used (see Section 5.1.114 [type], page 174).

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

```
int i=3;
i=i+1;        // overwriting
```

```
i;
↦ 4
int i=i+1;    // redefinition
↦ // ** redefining i **
i;
↦ 1
```

User defined names should start with a letter and consist of letters and digits only. As an exception of this rule, the character `@` may be used as part of a name, too. Capital and small letters are distinguished. Indexed names are built as a name followed by an int_expression in parentheses. A list of indexed names can be built as a name followed by an intvec_expression in parentheses.

```
ring R;
int n=3;
ideal j(3);
ideal j(n);     // is equivalent to the above
↦ // ** redefining j(3) **
ideal j(2)=x;
j(2..3);
↦ j(2)[1]=x j(3)[1]=0
```

Names may not coincide with reserved names (keywords). Type `reservedName();` to get a list of the reserved names. See Section 5.1.97 [reservedName], page 163. Names should not interfere with names of ring variables or, more generally, with monomials. See Section 6.4 [Identifier resolution], page 195.

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

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

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

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

3.4.4 Objects

Every object in SINGULAR has a type and a value. In most cases it has also a name and in some cases an attribute list. The value of an object may be examined simply by printing it with a print command: object;. The type of an object may be determined by means of the `typeof` function, the attributes by means of the `attrib` function (Section 5.1.115 [typeof], page 175, Section 5.1.1 [attrib], page 102):

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

Each object of type `poly`, `ideal`, `vector`, `module`, `map`, `matrix`, `number`, or `resolution` belongs to a specific ring. Also `list`, if at least one of the objects contained in the list belongs to a ring. These objects are local to the ring. Their names can be used for other objects in other rings. Objects from one ring can be mapped to another ring using maps or with the commands `fetch` or `imap`. See Section 4.8 [map], page 78, Section 5.1.28 [fetch], page 119, Section 5.1.41 [imap], page 127.

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

3.4.5 Type conversion and casting

Type conversion

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

1. `int` ↦ `ideal`
2. `poly` ↦ `ideal`
3. `intvec` ↦ `intmat`
4. `int` ↦ `intvec`
5. `int` ↦ `intvec`
6. `resolution` ↦ `list`
7. `ideal` ↦ `matrix`
8. `int` ↦ `matrix`
9. `intmat` ↦ `matrix`
10. `intvec` ↦ `matrix`

11. `module` ↦ `matrix`
12. `number` ↦ `matrix`
13. `poly` ↦ `matrix`
14. `vector` ↦ `matrix`
15. `ideal` ↦ `module`
16. `matrix` ↦ `module`
17. `vector` ↦ `module`
18. `int` ↦ `number`
19. `int` ↦ `poly`
20. `number` ↦ `poly`
21. `string` ↦ `proc`
22. `list` ↦ `resolution`
23. `int` ↦ `vector` (i ↦ i*`gen(1)`)
24. `poly` ↦ `vector` (p ↦ p*`gen(1)`)

Type casting

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

Possible type casts are:

1.	to `ideal`	from expression lists of `int`, `number`, `poly`
2.	to `ideal`	from `int`, `matrix`, `module`, `number`, `poly`, `vector`
3.	to `int`	from `number`, `poly`
4.	to `intvec`	from expression lists of `int`, `intmat`
5.	to `list`	from expression lists of any type
6.	to `matrix`	from `module`, `ideal`, `vector`, `matrix`. There are two forms to convert something to a matrix: if `matrix(` expression `)` is used then the size of the matrix is determined by the size of expression. But `matrix(` expression `,` m `,` n `)` may also be used - the result is a $m \times n$ matrix.
7.	to `module`	from expression lists of `int`, `number`, `poly`, `vector`
8.	to `module`	from `ideal`, `matrix`, `vector`
9.	to `number`	from `poly`
10.	to `poly`	from `int`, `number`
11.	to `string`	from `ideal`, `int`, `map`, `map`, `matrix`, `module`, `number`, `poly`, `proc`, `vector`

Example:

```
ring r=0,x,(c,dp);
number(3x);
↦ 0
number(poly(3));
↦ 3
ideal i=1,2,3,4,5,6;
print(matrix(i));
↦ 1,2,3,4,5,6
print(matrix(i,3,2));
↦ 1,2,
↦ 3,4,
```

```
↦ 5,6
vector v=[1,2];
print(matrix(v));
↦ 1,
↦ 2
module(matrix(i,3,2));
↦ _[1]=[1,3,5]
↦ _[2]=[2,4,6]
```

3.4.6 Flow control

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

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

Blocks are used whenever SINGULAR is used as a structured programming language. The `if` and `else` structures allow conditional execution of blocks (See Section 5.2.6 [if], page 181, Section 5.2.3 [else], page 180). `for` and `while` loops are available for repeated execution of blocks (See Section 5.2.5 [for], page 181, Section 5.2.10 [while], page 183). In procedure definitions the main part and the example section are blocks as well(See Section 4.13 [proc], page 92).

3.5 Input and output

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

Even though library loading may be considered an I/O operation, too, this section does not treat loading of libraries (see Section 5.1.55 [LIB], page 136).

Monitoring

A special form of I/O is monitoring. When monitoring is enabled, SINGULAR makes a typescript of everything printed on your terminal to a file. It is useful for students who need a hardcopy record of an interactive session as proof of an assignment. More generally, it is useful to create a protocol of a SINGULAR session. The `monitor` command enables and disables this feature (see Section 5.1.66 [monitor], page 143).

How to use links

Besides the usual I/O functions `write` and `read`, there are also the functions `dump` and `getdump` which store resp. retrieve the content of an entire SINGULAR session to resp. from a link. The `dump` and `getdump` commands are not available for DBM links.

For more information, see Section 5.1.121 [write], page 178, Section 5.1.93 [read], page 160, Section 5.1.20 [dump], page 114, Section 5.1.36 [getdump], page 124.

Example:

```
ring r; poly p = x+y;
dump("MPfile:w test.mp");   // dump the session to the file test.mp
kill r;                     // kill the basering
listvar();
↦ // LIB                  [0]  string standard.lib
getdump("MPfile:r test.mp");// read the dump from the file
listvar();
↦ // r                    [0]  *ring
↦ //      p                   [0]  poly
↦ // LIB                  [0]  string standard.lib
```

Specifying a link can be as easy as specifying a filename as a string. Except for MPtcp links, links even do not need to be explicitly opened or closed before resp. after they are used. To explicitly open or close a link, the `open` resp. `close` commands may be used (see Section 5.1.77 [open], page 150, Section 5.1.8 [close], page 107).

Links have various properties which can be checked for using the `status` function (see Section 5.1.105 [status], page 168).

Example:

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

ASCII links

Data that can be converted to a string can be written into files for storage or communication with other programs. The data is written in plain ASCII format. Reading from an ASCII link returns a string — conversion into other data is up to the user. This can be done, for example, using the command `execute` (see Section 5.1.23 [execute], page 117).

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

See Section 4.6.4 [ASCII links], page 67, for more information.

Example:

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

MPfile links

Data is stored in the binary MP format. Read and write access is very fast compared to ASCII links. All data (including such data that cannot be converted to a string) can be written to an MPfile link. Reading from an MPfile link returns the written expressions (i.e., not a string, in general).

MPfile links should primarily be used for storing large amounts of data (like dumps of the content of an entire SINGULAR session), and if the data to be stored cannot be easily converted from or to a string (like rings, or maps).

MPfile links are implemented on Unix-like operating systems only.

See Section 4.6.5.1 [MPfile links], page 69, for more information.

Example:

```
ring r;
// (over)write MPfile test.mp, link is specified as string
write("MPfile:w test.mp", x+y);
kill r;
def p = read("MPfile:r test.mp");
typeof(p); p;
↦ poly
↦ x+y
```

MPtcp links

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

MPtcp links should primarily be used for communications with other programs or for parallel computations (see, for example, Section A.27 [Parallelization with MPtcp links], page 237).

MPtcp links are implemented on Unix-like operating systems only.

See Section 4.6.5.2 [MPtcp links], page 70, for more information.

Example:

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

DBM links

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

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

See Section 4.6.6 [DBM links], page 73, for more information.

Example:

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

3.6 Procedures

Procedures contain sequences of SINGULAR. They are used to extend the set of commands with user defined commands. Procedures are defined by either typing them in on the command line or by loading them from a so-called library file with the Section 5.1.55 [LIB], page 136 command. Procedures are invoked like normal built-in commands, i.e., by typing their name followed by the list of arguments in parentheses. The invocation then executes the sequence of commands stored in the specified procedure. All defined procedures can be displayed by the command `listvar(proc);`.

3.6.1 Procedure definition

Syntax:
```
[static] proc proc_name [parameter_list]
["help_text"]
{
  procedure_body
}
[example
{
  sequence_of_commands;
}]
```

Purpose: defines a new function, the `proc` proc_name, with the additional information help_text, which is copied to the screen by `help proc_name;` and the `example` section which is executed by `example proc_name;`.
The help_text, the parameter_list, and the example section are optional. The default for a parameter_list is `(list #)`, see Section 3.6.3 [Parameter list], page 42. The help and example sections are ignored if the procedure is defined interactively, i.e., if it was not loaded from a file by a Section 5.1.55 [LIB], page 136 command.
Specifying `static` in front of the proc-definition (in a library file) makes this procedure local to the library, i.e., accessible only for the other procedures in the same library, but not for the users. So there is no reason anymore to define a procedure within another one (it just makes debugging harder).

Example of an interactive procedure definition

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

Example of a procedure definition in a library

First, the library definition:

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

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

Now, we load the library and execute the procedures defined there:

```
LIB "sample.lib";        // load the library sample.lib
example tab;             // show an example
↦ // proc tab from lib sample.lib
↦ EXAMPLE:
↦   for(int n=0; n<=4; n=n+1)
↦   { tab(4-n)+"*"+tab(n)+"+"+tab(n)+"*"; }
↦     *+*
↦    * + *
↦   *  +  *
↦  *   +   *
↦ *    +    *
↦
"*"+tab(3)+"*";          // use the procedure tab
↦ *   *
// the static procedure internal_tab is not accessible
"*"+internal_tab(3)+"*";
↦    ? 'sample.lib::internal_tab()' is a local procedure and cannot be
↦ accessed by an user.
↦    ? error occurred in Z line 5: `"*"+internal_tab(3)+"*";`
// show the help section for tab
help tab;
↦ // proc tab from lib sample.lib
↦ proc tab (int n)
↦
↦ "USAGE:    tab(n);  (n integer)
↦ RETURNS:  string of n space tabs
↦ EXAMPLE:  example tab; shows an example
```

```
↦
```

Guidelines for the help section of a procedure

The help text of a procedure should contain information about the usage, purpose, return values and generated objects. Particular assumptions or limitations should be listed. It should also be mentioned if global objects are generated or manipulated.

The libraries contained in the SINGULAR distribution use the following format for the help text:

```
USAGE:    <proc_name>(<parameters>);   <explanation of parameters>
[CREATE:  <description of created objects which are not returned>]
RETURN:   <description of the purpose and return value>
[NOTE:    <particular assumptions or limitations, details>]
EXAMPLE:  example <proc_name>; shows an example
```

3.6.2 Names in procedures

All variables are local to the procedure they are defined in. Locally defined variables cannot interfere with names in other procedures and are automatically deleted after leaving the procedure. Names can be made global by `export` (see Section 5.2.4 [export], page 180). These global names are not deleted automatically. Ring dependent variables are stored together with the ring and deleted when the ring is deleted.

Internally, local variables are stored using the nesting level. A variable is said to have nesting level 1, if it is local to a procedure that was called interactively, nesting level 2, if is local to a procedure that was called by a procedure of nesting level 1 etc. `listvar()` also displays the nesting level, nesting level 0 is used for global objects (see Section 5.1.58 [listvar], page 138). A ring may be 'moved up' by one nesting level with `keepring` (see Section 5.2.7 [keepring], page 182). All variables living in that ring are moved together with that ring.

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

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

3.6.3 Parameter list

Syntax: `( )`
`( parameter_definition )`

Purpose: defines the number, type and names of the arguments to a `proc`.
The parameter_list is optional. The default for a parameter_list is `(list #)` which means the arguments are referenced by `#[1], #[2]`, etc.

Example:

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

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

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

proc x3 (i,j)
{
... // can only be called with 2 arguments
    // of any type
    // (i,j) is the same as (def i,def j)
}

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

Note:

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

```
proc x6 (r)
{
... // this is correct even if the parameter is a ring
}

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

3.6.4 Procedure commands

See Section 5.2.4 [export], page 180; Section 5.2.7 [keepring], page 182; Section 5.2.9 [return], page 182.

3.7 Libraries

A library is a collection of SINGULAR procedures in a file.
SINGULAR reads a library with the command `LIB`, general information about the library is displayed by the command `help lib_name`. After loading the library, its procedures can be used like the built-in SINGULAR functions. To have the full functionality of a built-in function (like checking of the type of parameters, automatic loading of necessary other libraries, help pages etc.), libraries have to comply with the syntax rules described below.

Libraries can only be loaded with the `LIB` command:

Syntax: `LIB` string_expression`;`

Type: none

Purpose: reads a library of procedures from a file. If the given filename does not start with `.` or `/`, the following directories are searched for (in that order): the current directory, the directories given in the environment variable `SINGULARPATH`, some default directories relative to the location of the SINGULAR executable program, and finally some default absolute directories. You can view the search path which SINGULAR uses to locate its libraries, by starting up SINGULAR with the option `-v`, or by issuing the command `system("with");`".

Only the names of the procedures in the library are loaded, the body of the procedures is read during the first call of this procedure. This minimizes memory consumption by unused procedures. When SINGULAR is started with the `-q` or `--quiet` option, no message about the loading of a library is displayed. More exactly, option `-q` (and likewise `--quiet`) unsets option `loadLib` to inhibit monitoring of library loading (see Section 5.1.78 [option], page 150).

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

All loaded libraries are displayed by the `LIB` command without arguments:

Syntax: `LIB;`

Type: string

Purpose: shows all loaded libraries.

Example:

```
option(loadLib); // show loading of libraries;
                 // standard.lib is loaded
LIB;
↦ standard.lib
                 // the names of the procedures of inout.lib
LIB "inout.lib"; // are now known to Singular
↦ // ** loaded /home/obachman/Singular-1.2/Singular/LIB/inout.lib (1.6,
↦ 1998/05/14)
LIB;
↦ standard.lib,inout.lib
```

See Section 3.1.6 [Command line options], page 18; Section 2.3.3 [Procedures and libraries], page 10; Appendix D [SINGULAR libraries], page 250; Section 4.13 [proc], page 92; Section D.1 [standard_lib], page 250; Section 4.17 [string], page 96; Section 5.1.110 [system], page 172.

3.7.1 Format of a library

A library file can contain comments, an info- and version-string definition, `LIB` commands, `proc` commands and `proc` commands with `example` and `help` sections, i.e. the following keywords are allowed: `info`, `version`, `LIB`, `/* ... */`, `//`, `[static] proc`. Anything else is not recognized by the parser of SINGULAR and leads to an error message while loading the library. If an error occurs, loading is aborted and an error message is displayed, specifying the type of error and the line where it was detected.

The info- and version-string are defined as follows:

Syntax: `info` = string_constant ;

Purpose: defines the general help for the library. This text is displayed on `help lib_name;`.

Example:

```
info="
      This could be the general help of an library.
     Quote must be escaped with an \ such as \"
";
```

Note: In the info-string the characters `\` and `"` must be preceded by a `\` (escaped). It is recommended that the info string is placed on the top of a library file and contains general information about the library as well as a listing of all procedures available to the users (with a one line description of each procedure).

Syntax: `version` = string_constant ;

Purpose: defines the version number for the library. It is displayed when the library is loaded.

Example:

```
version="$Id: sample.lib,v 1.2 1998/05/07 singular Exp $";
  version="some version string";
```

Note: It is common practice to simply define the version string to be `"$Id:$"` and let a version control system expand it.

3.7.2 Guidelines for writing a library

Although there are very few enforced rules on how libraries are written (e.g., on whether or where the info- and version-string are defined), it is recommended that certain guidelines should be followed so that debugging and understanding are made easier.

1. The info- and version-string should appear at the beginning of the library, before procedure definitions.
2. The info-string should have the following format:

```
info="
      LIBRARY: <library_name> <one line description of the content>
      AUTHOR:  <name, affiliation, and email address of author>
      last modified: < date of last modification>

      <procedure1>;     <one line description of the purpose>
      .
      .
      <procedureN>;     <one line description of the purpose>";
```

The purpose of the one line procedure descriptions is not to give a short help for the procedure, but to help the user decide what procedure might be the right one for the job. Details can then be found in the help section of each procedure. Therefore parameters may be omitted or abbreviated if necessary.

3. Each procedure which is not declared `static` should have a help and example section as explained in Section 3.6.1 [Procedure definition], page 40.
4. Each procedure which should not be accessible by users should be declared `static`.
5. No procedures should be defined within the body of another procedure.
6. Executing procedures defined in a library should not result in any message displays, by default (please, shut up, by default). That is, only if special options or variable values are set, should a procedure display progress, debugging or any other information. A good way to accomplish this is the usage of the Section 5.1.12 [dbprint], page 110 command.

3.8 Debugging tools

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

3.8.1 Tracing of procedures

Setting the `TRACE` variable to 1 (resp. 3) results in a listing of the called procedures (resp. together with line numbers). If `TRACE` is set to 4, `Singular` displays each line before its interpretation and waits for the `RETURN` key being pressed. See Section 5.3.9 [TRACE var], page 188.

Example:

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

3.8.2 Break points

A break point can be put into a proc by inserting the command `~`. If `Singular` reaches a break point it asks for lines of commands (line-length must be less than 80 characters) from the user. It returns to normal execution if given an empty line. See Section 5.2.11 [~], page 184.

Example:

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

↦ 3
```

3.8.3 Printing of data

The procedure `dbprint` is useful for optional output of data: it takes 2 arguments and prints the second argument, if the first argument is positive; it does nothing otherwise. See Section 5.1.12 [dbprint], page 110; Section 5.3.11 [voice], page 189.

3.8.4 libparse

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

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

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

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

The following syntax checks are performed in any case:

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

Its output lists all procedures that have been parsed successfully:

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

where the following abbreviations are used:

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

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

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

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

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

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

4 Data types

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

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

For the general syntax of a declaration see Section 3.4.1 [General command syntax], page 29.

4.1 def

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

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

```
ring r=32003,(x,y,z),dp;
poly f = x;
def s=basering;
setring s;
nameof(basering);
↦ s
listvar();
↦ // s                    [0]  *ring
↦ //      f                   [0]  poly
↦ // r                    [0]  ring
↦ // LIB                  [0]  string standard.lib
poly g = y;
kill f;
listvar(r);
↦ // r                    [0]  ring
↦ // g                    [0]  poly
```

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

4.1.1 def declarations

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

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

Default: none

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

Example:

```
def i=2;
typeof(i);
↦ int
```

See Section 5.1.115 [typeof], page 175.

4.2 ideal

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

Note: `size` counts only the non zero generators of an ideal whereas `ncols` counts all generators.

4.2.1 ideal declarations

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

Purpose: defines an ideal.

Default: 0

Example:

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

4.2.2 ideal expressions

An ideal expression is:

1. an expression list of poly expressions and ideal expressions

2. an identifier of type ideal
3. a function returning ideal
4. ideal expressions combined by the arithmetic operations + or *
5. a power of an ideal expression (operator ^ or **)
   Note that the computation of the product i*i involves all products of generators of i while i^2 involves only the different ones, and is therefore faster.
6. a type cast to ideal

Example:

```
ring r=0,(x,y,z),dp;
ideal m = maxideal(1);
m;
↦ m[1]=x
↦ m[2]=y
↦ m[3]=z
poly f = x2;
poly g = y3;
ideal i = x*y*z , f-g, g*(x-y) + f^4 ,0, 2x-z2y;
ideal M = maxideal(10);
timer =0;
i = M*M;
timer;
↦ 1
ncols(i);
↦ 231
timer =0;
i = M^2;
ncols(i);
↦ 231
timer;
↦ 0
i[ncols(i)];
↦ x20
vector v = [x,y-z,x2,y-x,x2yz2-y];
ideal j = ideal(v);
```

4.2.3 ideal operations

+ addition (concatenation of the generators and simplification)

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

^ exponentiation (by a non-negative integer)

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

Note: For simplification of an ideal, see also Section 5.1.101 [simplify], page 165.

Example:

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

4.2.4 ideal related functions

`char_series`
: irreducible characteristic series (see Section 5.1.5 [char_series], page 106)

`coeffs` matrix of coefficients (see Section 5.1.10 [coeffs], page 108)

`contract` contraction by an ideal (see Section 5.1.11 [contract], page 110)

`diff` partial derivative (see Section 5.1.18 [diff], page 113)

`degree` multiplicity, dimension and codimension of the ideal of leading terms (see Section 5.1.15 [degree], page 112)

`dim` Krull dimension of basering modulo the ideal of leading terms (see Section 5.1.19 [dim], page 114)

`eliminate`
: elimination of variables (see Section 5.1.21 [eliminate], page 115)

`facstd` factorizing Groebner basis algorithm (see Section 5.1.26 [facstd], page 118)

`factorize`
: ideal of factors of a polynomial (see Section 5.1.27 [factorize], page 119)

`fglm` Groebner basis computation from a Groebner basis w.r.t. a different ordering (see Section 5.1.29 [fglm], page 121)

`finduni` computation of univariate polynomials lying in a zero dimensional ideal (see Section 5.1.32 [finduni], page 122)

`groebner` Groebner basis computation (a wrapper around `std,stdhilb,stdfglm`,...) (see Section 5.1.37 [groebner], page 124)

`homog` homogenization with respect to a variable (see Section 5.1.40 [homog], page 126)

`hilb` Hilbert series of a standard basis (see Section 5.1.39 [hilb], page 125)

`indepSet` sets of independent variables of an ideal (see Section 5.1.42 [indepSet], page 127)

`interred` interreduction of an ideal (see Section 5.1.44 [interred], page 129)

`intersect`
: ideal intersection (see Section 5.1.45 [intersect], page 130)

`jacob` ideal of all partial derivatives resp. jacobian matrix (see Section 5.1.46 [jacob], page 130)

`jet` Taylor series up to a given order (see Section 5.1.47 [jet], page 131)

`kbase` vector space basis of basering modulo ideal of leading terms (see Section 5.1.48 [kbase], page 132)

`koszul` Koszul matrix (see Section 5.1.51 [koszul], page 134)

`lead` leading terms of a set of generators (see Section 5.1.52 [lead], page 134)

`lift` lift-matrix (see Section 5.1.56 [lift], page 137)

`liftstd` standard basis and transformation matrix computation (see Section 5.1.57 [liftstd], page 137)

`lres` minimal resolution for homogeneous ideals (see Section 5.1.59 [lres], page 139)

`maxideal` power of the maximal ideal at 0 (see Section 5.1.60 [maxideal], page 140)

`minbase` minimal generating set of a homogeneous ideal resp. module or an ideal resp. module in a local ring (see Section 5.1.62 [minbase], page 141)

`minor` set of minors of a matrix (see Section 5.1.63 [minor], page 141)

`modulo` represents $(h1 + h2)/h1 \cong h2/(h1 \cap h2)$ (see Section 5.1.65 [modulo], page 143)

`mres` minimal free resolution of an ideal resp. module, also minimizing the given ideal resp. module (see Section 5.1.67 [mres], page 144)

`mstd` standard basis and minimal generating set of an ideal (see Section 5.1.68 [mstd], page 145)

`mult` multiplicity resp. degree of the ideal of leading terms (see Section 5.1.69 [mult], page 145)

`ncols` number of columns (see Section 5.1.72 [ncols], page 147)

`preimage` preimage under a ring map (see Section 5.1.85 [preimage], page 155)

`qhweight` quasihomogeneous weights of an ideal (see Section 5.1.89 [qhweight], page 157)

`quotient` ideal quotient (see Section 5.1.91 [quotient], page 158)

`reduce` normalform with respect to a standard basis (see Section 5.1.94 [reduce], page 161)

`res` minimal free resolution of an ideal resp. module but not changing the given ideal resp. module (see Section 5.1.96 [res], page 162)

`simplify` simplify a set of polynomials (see Section 5.1.101 [simplify], page 165)

`size` number of non-zero generators (see Section 5.1.102 [size], page 166)

`sortvec` permutation for sorting ideals resp. modules (see Section 5.1.103 [sortvec], page 167)

`sres` free resolution of a standard basis (see Section 5.1.104 [sres], page 167)

`std` standard basis computation (see Section 5.1.106 [std], page 169)

`stdfglm` standard basis computation with fglm technique (see Section 5.1.107 [stdfglm], page 170)

`stdhilb` Hilbert driven standard basis computation (see Section 5.1.108 [stdhilb], page 171

`subst` substitute a ring variable (see Section 5.1.109 [subst], page 171)

`syz` computation of the first syzygy module (see Section 5.1.111 [syz], page 173)

`vdim` vector space dimension of basering modulo ideal of leading terms (see Section 5.1.118 [vdim], page 176)

`weight` optimal weights (see Section 5.1.120 [weight], page 177)

4.3 int

Variables of type int represent the machine integers and are, therefore, limited in their range (e.g., the range is between -2147483647 and 2147483647 on 32-bit machines). They are mainly used to count things (dimension, rank, etc.), in loops (see Section 5.2.5 [for], page 181), and to represent boolean values (FALSE is represented by 0, every other value means TRUE, see Section 4.3.5 [boolean expressions], page 58).

Integers consist of a sequence of digits, possibly preceded by a sign. A space is considered as a separator, so it is not allowed between digits. A sequence of digits outside the allowed range is converted to the type `number` if possible.

4.3.1 int declarations

Syntax: `int` name = int_expression ;

Purpose: defines an integer variable.

Default: 0

Example:

```
int i = 42;
int j = i + 3; j;
↦ 45
i = i * 3 - j; i;
↦ 81
int k;   // assigning the default value 0 to k
k;
↦ 0
```

4.3.2 int expressions

An int expression is:

1. a sequence of digits (if the number represented by this sequence is too large to fit into the range of integers it is automatically converted to the type number, if a basering is defined)
2. an identifier of type int
3. a function returning int
4. int expressions combined by the arithmetic operations `+`, `-`, `*`, `div`, `/`, `%` (`mod`), or `^`
5. a boolean expression
6. a type cast to int

Note: Variables of type int represent the compiler integers and are, therefore, limited in their range (see Section 6.1 [Limitations], page 190). If this range is too small the expression must be converted to the type number over a ring with characteristic 0.

Example:

```
12345678901;
↦    ? `12345678901` greater than 2147483647(max. integer representatio
↦ n)
↦    ? error occurred in Z line 1: `12345678901;`
typeof(_);
↦ none
ring r=0,x,dp;
12345678901;
↦ 12345678901
typeof(_);
↦ number
// Note: 11*13*17*100*200*2000*503*1111*222222
// returns a machine integer:
```

```
11*13*17*100*200*2000*503*1111*222222;
↦ // ** int overflow(*), result may be wrong
↦ // ** int overflow(*), result may be wrong
↦ // ** int overflow(*), result may be wrong
↦ // ** int overflow(*), result may be wrong
↦ -1875651584
// using the type cast number for a greater allowed range
number(11)*13*17*100*200*2000*503*1111*222222;
↦ 12075748128684240000000
ring rp=32003,x,dp;
12345678901;
↦ 9603
typeof(_);
↦ number
intmat m[2][2] = 1,2,3,4;
m;
↦ 1,2,
↦ 3,4
m[2,2];
↦ 4
typeof(_);
↦ int
det(m);
↦ -2
m[1,1] + m[2,1] == trace(m);
↦ 0
! 0;
↦ 1
1 and 2;
↦ 1
intvec v = 1,2,3;
def d =transpose(v)*v;    // scalarproduct gives an 1x1 intvec
typeof(d);
↦ intvec
int i = d[1];             // access the first (the only) entry in the intvec
ring rr=31,(x,y,z),dp;
poly f = 1;
i = int(f);               // cast to int
// Integers may be converted to constant  polynomials by an assignment,
poly g=37;
// define the constant polynomial g equal to the image of
// the integer j in the actual coefficient field, here it equals 6
g;
↦ 6
```

See Section 4.11 [number], page 86; Section 3.4.5 [Type conversion and casting], page 34.

4.3.3 int operations

++ changes its operand to its successor, is itself no int expression

`--` changes its operand to its predecessor, is itself no int expression

`+` addition

`-` negation or subtraction

`*` multiplication

`div`, `/` integer division (omitting the remainder)

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

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

`<`, `>`, `<=`, `>=`, `==`, `<>`
comparison

Note: An assignment `j=i++;` or `j=i--;` is not allowed, in particular it does not change the value of `j`, see Section 6.1 [Limitations], page 190.

Note: `/` might no longer be available in the future.

Example:

```
int i=1;
int j;
i++; i;  i--; i;
↦ 2
↦ 1
// ++ and -- do not return a value as in C, the variable j is unchanged
j = i++; j; i;
↦    ? right side is not a datum
↦    ? error occurred in Z line 5: ‘j = i++; j; i;‘
↦ 0
↦ 2
j = i--; j; i;
↦    ? right side is not a datum
↦    ? error occurred in Z line 6: ‘j = i--; j; i;‘
↦ 0
↦ 1
i+2, 2-i, 5^2;
↦ 3 1 25
5 div 2, 8%3;
↦ 2 2
1<2, 2<=2;
↦ 1 1
```

4.3.4 int related functions

`char` characteristic of the coefficient field of a ring (see Section 5.1.4 [char], page 105)

`deg` degree of a poly resp. vector (see Section 5.1.14 [deg], page 111)

`det` determinant (see Section 5.1.17 [det], page 113)

`dim` Krull dimension of basering modulo ideal resp. module of leading terms (see Section 5.1.19 [dim], page 114)

`extgcd` Bezout representation of gcd (see Section 5.1.25 [extgcd], page 117)

`find` position of a substring in a string (see Section 5.1.31 [find], page 121)

`gcd` greatest common divisor (see Section 5.1.34 [gcd], page 123)

`koszul` Koszul matrix (see Section 5.1.51 [koszul], page 134)

`memory` memory usage (see Section 5.1.61 [memory], page 141)

`mult` multiplicity of an ideal resp. module of leading terms (see Section 5.1.69 [mult], page 145)

`ncols` number of columns (see Section 5.1.72 [ncols], page 147)

`npars` number of ring parameters (see Section 5.1.73 [npars], page 148)

`nrows` number of rows of a matrix resp. the rank of the free module where the vector or module lives (see Section 5.1.75 [nrows], page 149)

`nvars` number of ring variables (see Section 5.1.76 [nvars], page 150)

`ord` degree of the leading term of a poly resp. vector (see Section 5.1.79 [ord], page 153)

`par` n-th paramter of the basering (see Section 5.1.81 [par], page 154)

`pardeg` degree of a number considered as a polynomial in the ring parameters (see Section 5.1.82 [pardeg], page 154)

`prime` the next lower prime (see Section 5.1.86 [prime], page 156)

`random` a pseudo random integer between the given limits (see Section 5.1.92 [random], page 159)

`regularity`
regularity of a resolution (see Section 5.1.95 [regularity], page 161)

`rvar` test, if the given expression or string is a ring variable (see Section 5.1.99 [rvar], page 164)

`size` number of elements in an object (see Section 5.1.102 [size], page 166)

`trace` trace of an integer matrix (see Section 5.1.112 [trace], page 174)

`var` n-th ring variable of the basering (see Section 5.1.116 [var], page 176)

`vdim` vector space dimension of basering modulo ideal resp. of freemodule modulo module of leading terms (see Section 5.1.118 [vdim], page 176)

4.3.5 boolean expressions

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*)

The following is the list of available comparisons of objects of a same type.

Note: There are no comparisons for ideals and modules, resolution and maps.

1. an integer comparison:

```
i == j
i != j    // or     i <> j
i <= j
i >= j
i > j
i < j
```

2. a number comparison:

```
m == n
m != n    // or     m <> n
m < n
m > n
m <= n
m >= n
```

   For numbers from Z/p or from field extensions not all operations are useful:
   - 0 ist always the smallest element,
   - in Z/p the representatives in the range -(p-1)/2..(p-1)/2 when p>2 resp. 0 and 1 for p=2 are used for comparisons,
   - in field extensions the last two operations (`>=,<=`) yield always TRUE (1) and the `<` and `>` are equivalent to `!=`.

3. a polynomial or vector comparison:

```
f == g
f != g    // or     f <> g
f <= g    // comparing the leading term w.r.t. the monomial order
f <  g
f >= g
f >  g
```

4. an intmat or matrix comparison:

```
v == w
v != w    // or     v <> w
```

5. an intvec or string comparison:

```
f == g
f != g    // or     f <> g
f <= g    // comparing lexicographically
f >= g    // w.r.t. the order specified by ASCII
f >  g
f <  g
```

6. boolean expressions combined by boolean operations (`and`, `or`, `not`)

   Note: All arguments of a logical expressions are first evaluated and then the value of the logical expression is determined. For example, the logical expressions `(a || b)` is evaluated by first evaluating `a` *and* `b`, even though the value of `b` has no influence on the value of `(a || b)`, if `a` evaluates to true.

   Note that this evaluation is different from the left-to-right, conditional evaluation of logical expressions (as found in most programming languages). For example, in these other languages, the value of `(1 || b)` is determined without ever evaluating `b`.

   See Section 6.2 [Major differences to the C programming language], page 190.

4.3.6 boolean operations

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

The precedence of the boolean operations is:

1. parentheses
2. comparisons
3. not
4. and
5. or

Example:

```
(1>2) and 3;
↦ 0
1 > 2 and 3;
↦ 0
! 0 or 1;
↦ 1
!(0 or 1);
↦ 0
```

4.4 intmat

Integer matrices are matrices with integer entries. For the range of integers see Section 6.1 [Limitations], page 190. Integer matrices do not belong to a ring, they may be defined without a basering being defined. An intmat can be multiplied by and added to an int; in this case the int is converted into an intmat of the right size with the integer on the diagonal. The integer `1`, for example, is converted into the unit matrix.

4.4.1 intmat declarations

Syntax: `intmat` name = intmat_expression ;
`intmat` name [rows] [cols] = intmat_expression ;
`intmat` name [rows] [cols] = list_of_int_expressions ;
rows and cols must be int expressions.

Purpose: defines an intmat variable.
Given a list of integers, the matrix is filled up with the first row from the left to the right, then the second row and so on. If the int_list contains less than rows*cols elements, the matrix is filled up with zeros; if it contains more elements, only the first rows*cols elements are used.

Default: 0 (1 x 1 matrix)

Example:

```
intmat im[3][5]=1,3,5,7,8,9,10,11,12,13;
im;
↦ 1,3,5,7,8,
↦ 9,10,11,12,13,
↦ 0,0,0,0,0
im[3,2];
↦ 0
intmat m[2][3] = im[1..2,3..5];  // defines a submatrix
m;
↦ 5,7,8,
↦ 11,12,13
```

4.4.2 intmat expressions

An intmat expression is:

1. a list of int expressions, intvec expressions, or intmat expressions
2. an identifier of type intmat
3. a function returning intmat
4. intmat operations with int (+, -, *, div, %)
5. intmat operations (+, -, *)
6. a type cast to intmat

Example:

```
intmat Idm[3][3];
Idm +1;          // add the unit intmat
↦ 1,0,0,
↦ 0,1,0,
↦ 0,0,1
intmat m1[3][2] = _,1,-2;  // take entries from the last result
m1;
↦ 1,0,
↦ 0,0,
↦ 1,0
intmat m2[2][3]=1,0,2,4,5,1;
transpose(m2);
↦ 1,4,
↦ 0,5,
↦ 2,1
intvec v1=1,2,4;
intvec v2=5,7,8;
m1=v1,v2;         // fill m1 with v1 and v2
m1;
↦ 1,2,
↦ 4,5,
↦ 7,8
trace(m1*m2);
```

```
↦ 56
```

See Section 4.11 [number], page 86; Section 3.4.5 [Type conversion and casting], page 34.

4.4.3 intmat operations

`+`	addition with intmat or int; the int is converted into a diagonal intmat
`-`	negation or subtraction with intmat or int; the int is converted into a diagonal intmat
`*`	multiplication with intmat, intvec, or int; the int is converted into a diagonal intmat
`div,/`	division of entries in the integers (omitting the remainder)
`%, mod`	entries modulo int (remainder of the division)
`<>, ==`	comparison
intmat_expression `[` intvec_expressions`,` intvec_expression `]`	is an intmat entry, where the first index indicates the row and the second the column

Example:

```
intmat m[2][4] = 1,0,2,4,0,1,-1,0,3,2,1,-2;
m;
↦ 1,0,2,4,
↦ 0,1,-1,0
m[2,3];          // entry at row 2, col 3
↦ -1
size(m);         // number of entries
↦ 8
intvec v = 1,0,-1,2;
m * v;
↦ 7,1
typeof(_);
↦ intvec
intmat m1[4][3] = 0,1,2,3,v,1;
intmat m2 = m * m1;
m2;              //  2 x 3 intmat
↦ -2,5,4,
↦ 4,-1,-1
m2*10;           // multiply each entry of m with 10;
↦ -20,50,40,
↦ 40,-10,-10
-m2;
↦ 2,-5,-4,
↦ -4,1,1
m2 % 2;
↦ 0,1,0,
↦ 0,1,1
m2 div 2;
↦ -1,2,2,
```

```
↦ 2,0,0
m2[2,1];          // entry at row 2, col 1
↦ 4
m1[2..3,2..3];   // submatrix
↦ 1 0 2 1
m2[nrows(m2),ncols(m2)];      // the last entry of intmat m2
↦ -1
```

4.4.4 intmat related functions

`betti`	Betti numbers of a free resolution (see Section 5.1.3 [betti], page 104)
`det`	determinant (see Section 5.1.17 [det], page 113)
`ncols`	number of cols (see Section 5.1.72 [ncols], page 147)
`nrows`	number of rows (see Section 5.1.75 [nrows], page 149)
`random`	create a pseudo random intmat (see Section 5.1.92 [random], page 159)
`size`	total number of entries (see Section 5.1.102 [size], page 166)
`transpose`	transpose of an intmat (see Section 5.1.113 [transpose], page 174)
`trace`	trace of an intmat (see Section 5.1.112 [trace], page 174)

4.5 intvec

Variables of type intvec are lists of integers. For the range of integers see Section 6.1 [Limitations], page 190. They may be used for simulating integers sets (and other sets if the intvec is used as an index set for other objects). Addition and subtraction of an intvec with in int is done element wise.

4.5.1 intvec declarations

Syntax: `intvec` name = intvec_expression ;

Purpose: defines an intvec variable.
An intvec consists of an ordered list of integers.

Default: 0

Example:

```
intvec iv=1,3,5,7,8;
iv;
↦ 1,3,5,7,8
iv[4];
↦ 7
iv[3..size (iv)];
↦ 5 7 8
```

4.5.2 intvec expressions

An intvec expression is:

1. a list of int expressions and of intvec expressions
2. a range: int expression .. int expression
3. a function returning intvec
4. intvec operations with int (+, -, *, /, %)
5. intvec operations (+, -)
6. intvec operation with intmat (*)
7. a type cast to intvec

Example:

```
intvec v=-1,2;
intvec w=v,v;          // concatenation
w;
↦ -1,2,-1,2
w = -2..2,v,1;
w;
↦ -2,-1,0,1,2,-1,2,1
intmat m[3][2] = 0,1,2,-2,3,1;
m*v;
↦ 2,-6,-1
typeof(_);
↦ intvec
v = intvec(m);
v;
↦ 0,1,2,-2,3,1
ring r;
poly f = x2z + 2xy-z;
f;
↦ x2z+2xy-z
v = leadexp(f);
v;
↦ 2,0,1
```

4.5.3 intvec operations

+	addition with intvec or int (component-wise)
-	negation or subtraction with intvec or int (component-wise)
*	multiplication with int (component-wise)
/, div	division with int (component-wise)
%, mod	modulo (component-wise)
<>, ==, <=, >=, >, <	comparison (done lexicographically)

intvec_expression [int_expression]
: is an element of the intvec; the first element has index one.

Example:

```
intvec iv =  1,3,5,7,8;
iv+1;               // add 1 to each entry
↦ 2,4,6,8,9
iv*2;
↦ 2,6,10,14,16
iv;
↦ 1,3,5,7,8
iv-10;
↦ -9,-7,-5,-3,-2
iv=iv,0;
iv;
↦ 1,3,5,7,8,0
iv div 2;
↦ 0,1,2,3,4,0
iv+iv;              // componentwise addition
↦ 2,6,10,14,16,0
iv[size(iv)-1];     // last-1 entry
↦ 8
intvec iw=2,3,4,0;
iv==iw;             // lexicographic comparision
↦ 0
iv < iw;
↦ 1
iv != iw;
↦ 1
iv[2];
↦ 3
iw = 4,1,2;
iv[iw];
↦ 7 1 3
```

4.5.4 intvec related functions

`hilb`	returns Hilbert series as intvec (see Section 5.1.39 [hilb], page 125)
`indepSet`	sets of independent variables of an ideal (see Section 5.1.42 [indepSet], page 127)
`leadexp`	the exponent vector of the leading monomial (see Section 5.1.54 [leadexp], page 135)
`nrows`	number of rows (see Section 5.1.75 [nrows], page 149)
`qhweight`	returns quasihgomogeneous weights (see Section 5.1.89 [qhweight], page 157)
`size`	length of the intvec (see Section 5.1.102 [size], page 166)
`sortvec`	permutation for sorting ideals/modules (see Section 5.1.103 [sortvec], page 167)

`transpose`
: transpose of an intvec, returns an intmat (see Section 5.1.113 [transpose], page 174)

`weight`
: returns weights for the weighted ecart method (see Section 5.1.120 [weight], page 177)

4.6 link

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

- ASCII links (see Section 4.6.4 [ASCII links], page 67)
- MPfile links (see Section 4.6.5.1 [MPfile links], page 69)
- MPtcp links (see Section 4.6.5.2 [MPtcp links], page 70)
- DBM links (see Section 4.6.6 [DBM links], page 73)

4.6.1 link declarations

Syntax: `link` name = string_expression ;

Purpose: defines a new communication link.

Default: none

Example:

```
link l=":w example.txt";
int i=22;          // cf. ASCII links for explanation
string s="An int follows:";
write(l,s,i);
l;
↦ // type : ASCII
↦ // mode : w
↦ // name : example.txt
↦ // open : yes
↦ // read : not ready
↦ // write: ready
close(l);          //
read(l);
↦ An int follows:
↦ 22
↦
close(l);
```

4.6.2 link expressions

A link expression is:

1. an identifier of type link
2. a string describing the link

A link is described by a string which consists of two parts: a property string followed by a name string. The property string describes the type of the link (`ASCII`, `MPfile`, `MPtcp` or `DBM`) and the mode of the link (e.g., open for read, write or append). The name string describes the filename of the link, resp. a network connection for MPtcp links.

For a detailed format description of the link describing string see:

- for ASCII links: Section 4.6.4 [ASCII links], page 67
- for MPfile links: Section 4.6.5.1 [MPfile links], page 69
- for MPtcp links: Section 4.6.5.2 [MPtcp links], page 70
- for DBM links: Section 4.6.6 [DBM links], page 73

4.6.3 link related functions

`close`	closes a link (see Section 5.1.8 [close], page 107)
`dump`	generates a dump of all variables and their values (see Section 5.1.20 [dump], page 114)
`getdump`	reads a dump (see Section 5.1.36 [getdump], page 124)
`open`	opens a link (see Section 5.1.77 [open], page 150)
`read`	reads from a link (see Section 5.1.93 [read], page 160)
`status`	gets the status of a link (see Section 5.1.105 [status], page 168)
`write`	writes to a link (see Section 5.1.121 [write], page 178)
`kill`	closes and kills a link (see Section 5.1.49 [kill], page 133)

4.6.4 ASCII links

Via ASCII links data that can be converted to a string can be written into files for storage or communication with other programs. The data is written in plain ASCII format. The output format of polynomials is done w.r.t. the value of the global variable `short` (see Section 5.3.7 [short], page 187). Reading from an ASCII link returns a string — conversion into other data is up to the user. This can be done, for example, using the command `execute` (see Section 5.1.23 [execute], page 117).

The ASCII link describing string has to be one of the following:

1. `"ASCII: " +` filename
   the mode (read or append) is set by the first `read` or `write` command.
2. `"ASCII:r " +` filename
   opens the file for reading.
3. `"ASCII:w " +` filename
   opens the file for overwriting.
4. `"ASCII:a " +` filename
   opens the file for appending.

There are the following default values:

- the type `ASCII` may be omitted since ASCII links are the default links.
- if non of `r`, `w`, or `a` is specified, the mode of the link is set by the first `read` or `write` command on the link. If the first command is `write`, the mode is set to `a` (append mode).
- if the filename is omitted, `read` reads from stdin and `write` writes to stdout.

Using these default rules, the string `":r temp"` describes a link which is equivalent to the link `"ASCII:r temp"`: an ASCII link to the file `temp` which is opened for reading. The string `"temp"` describes an ASCII link to the file `temp`, where the mode is set by the first `read` or `write` command. See also the example below.

Note that the filename may contain a path. An ASCII link can be used either for reading or for writing, but not for both at the same time. A `close` command must be used before a change of I/O direction. Types without a conversion to `string` cannot be written.

Example:

```
ring r=32003,(x,y,z),dp;
link l=":w example.txt";     // type is ASCII, mode is overwrite
l;
↦ // type : ASCII
↦ // mode : w
↦ // name : example.txt
↦ // open : no
↦ // read : not ready
↦ // write: not ready
status(l, "open", "yes");    // link is not yet opened
↦ 0
ideal i=x2,y2,z2;
write (l,1,";",2,";","ideal i=",i,";");
status(l, "open", "yes");    // now link is open
↦ 1
status(l, "mode");           // for writing
↦ w
close(l);                    // link is closed
write("example.txt","int j=5;");// data is appended to file
read("example.txt");         // data is returned as string
↦ 1
↦ ;
↦ 2
↦ ;
↦ ideal i=
↦ x2,
↦ y2,
↦ z2
↦ ;
↦ int j=5;
↦
execute read(l);             // read string is executed
↦ 1
↦ 2
```

```
↦ // ** redefining i **
close(l);                    // link is closed
```

4.6.5 MP links

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. MP links can be established using files (link type is `MPfile`) or using TCP sockets (link type is `MPtcp`). All data (including such data that can not be converted to a string) can be written to an MP link. For ring-dependent data, a ring description is written together with the data. Reading from an MP link returns an expression (not a string) which was evaluated after the read operation. If the expression read from an MP link is not from the same ring as the current ring, then a `read` changes the current ring.

Currently, MP links are only available on Unix platforms and data is written without attributes (which is likely to change in future versions). For a general description of MP, see `http://symbolicnet.mcs.kent.edu/areas/mp.html`.

4.6.5.1 MPfile links

MPfile links provide the possibility to store data in a file using the binary MP format. Read and write operations are very fast compared to ASCII links. Therefore, for storing large amounts of data, MPfile links should be used, instead of ASCII links. Unlike ASCII links, data read from MPfile links is returned as expressions one at a time, and not as a string containing the entire content of the file. Furthermore, ring dependent data is stored together with a ring description. Therefore, reading ring-dependent data might change the current ring.

The MPfile link describing string has to be one of the following:

1. `"MPfile: " +` filename
   the mode (read or append) is set by the first `read` or `write` command.
2. `"MPfile:r " +` filename
   opens the file for reading.
3. `"MPfile:w " +` filename
   opens the file for overwriting.
4. `"MPfile:a " +` filename
   opens the file for appending.

There are the following default values:

- if none of `r`, `w`, or `a` is specified, the mode of the link is set by the first `read` or `write` command on the link. If the first command is `write`, the mode is set to `a` (append mode).

Note that the filename may contain a path. An MP file link can be used either for reading or for writing, but not for both at the same time. A `close` command must be used before a change of I/O direction.

Example:

```
ring r;
link l="MPfile:w example.mp"; // type=MPfile, mode=overwrite
l;
↦ // type : MPfile
↦ // mode : w
↦ // name : example.mp
↦ // open : no
↦ // read : not ready
↦ // write: not ready
ideal i=x2,y2,z2;
write (l,1, i, "hello world");// write three expressions
write(l,4);                   // append one more expression
close(l);                     // link is closed
// open the file for reading now
read(l);                      // only first expression is read
↦ 1
kill r;                       // no basering active now
def i = read(l);              // second expression
// notice that current ring was set, the name was assigned
// automatically
listvar(ring);
↦ // mpsr_r0              [0]  *ring
def s = read(l);              // third expression
listvar();
↦ // s                     [0]  string hello world
↦ // mpsr_r0              [0]  *ring
↦ //      i                    [0]  ideal, 3 generator(s)
↦ // l                     [0]  link
↦ // LIB                   [0]  string standard.lib
close(l);                     // link is closed
dump("MPfile:w example.mp");  // dump everything to example.mp
kill i, s;                    // kill i and s
getdump("MPfile: example.mp");// get previous dump
listvar();                    // got all variables and values back
↦ // mpsr_r0              [0]  *ring
↦ //      i                    [0]  ideal, 3 generator(s)
↦ // s                     [0]  string hello world
↦ // l                     [0]  link
↦ // LIB                   [0]  string standard.lib
```

4.6.5.2 MPtcp 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. MPtcp links can be opened in four different modes:

`listen` SINGULAR acts as a server.

`connect` SINGULAR acts as a client.

`launch` SINGULAR acts as a client, launching an application as server.

`fork` SINGULAR acts as a client, forking another SINGULAR as server.

The MPtcp link describing string has to be

- listen mode:
  1. `"MPtcp:listen --MPport " +` portnumber

  SINGULAR becomes a server and waits at the port for a connect call.
- connect mode:
  2. `"MPtcp:connect --MPport " +` portnumber
  3. `"MPtcp:connect --MPhost" +` hostname `+ " --MPport " +` portnumber

  SINGULAR becomes a client and connects to a server waiting at the host and port.
- launch mode:
  4. `"MPtcp:launch"`
  5. `"MPtcp:launch --MPapplication " +` application
  6. `"MPtcp:launch --MPhost " +` hostname
  7. `"MPtcp:launch --MPhost " +` hostname `+ " --MPapplication " +` application

  SINGULAR becomes a client and starts the application on a different host which acts as a server.
- fork mode:
  8. `"MPtcp:fork"`

  SINGULAR becomes a client and forks another SINGULAR on the same host which acts as a server.

There are the following default values:

- if none of `listen`, `connect`, `launch` or `fork` is specified, the default mode is set to `fork`.
- if no application is specified (in mode `launch`) the default application is the value of `system("Singular") + " -b"`. (This evaluates to the absolute path of the SINGULAR currently running with the option "-b" appended.
- if no hostname is specified the local host is used as default host.

To open an MPtcp link in launch mode, the application to launch must either be given with an absolute pathname, or must be in a directory contained in the search path. The launched application acts as a server, whereas the SINGULAR that actually opened the link acts as a client. SINGULAR automatically appends the command line arguments `"--MPmode connect --MPhost` hostname `--MPport` portnumber" to the command line of the server application. Both hostname and portnumber are substituted by the values from the link specification. The client "listens" at the given port until the server application does a connect call. If SINGULAR is used as server application it has to be started with the command line option `-b`. Since launching is done using `rsh`, the host on which the application should run must have an entry in the `.rhosts` file. Even the local machine must have an entry if applications should be launched locally.

If the MPtcp link is opened in fork mode a child of the current SINGULAR is forked. All variables and their values are inherited by the child. The child acts as a server whereas the SINGULAR that actually opened the link acts as a client.

To arrange the evaluation of an expression by a server, the expression must be quoted using the command `quote` (see Section 5.1.90 [quote], page 158), so that a local evaluation is prevented.

Otherwise, the expression is evaluated first, and the result of the evaluation is written, instead of the expression which is to be evaluated.

If SINGULAR is in server mode, the value of the variable `mp_ll` is the MPtcp link connecting to the client and SINGULAR is in an infinite read-eval-write loop until the connection is closed from the client side (by closing its connecting link). Reading and writing is done to the link `mp_ll`: After an expression is read, it is evaluated and the result of the evaluation is written back. That is, for each expression which was written to the server, there is exactly one expression written back. This might be an "empty" expression, if the evaluation on the server side does not return a value.

MPtcp links should explicitly be opened before being used. MPtcp links are bidirectional, i.e., can be be used for both, writing and reading. Reading from an MPtcp link blocks until data was written to that link. The `status` command can be used to check whether there is data to read.

Example:

```
LIB "general.lib"; // needed for "killall" command
link l="MPtcp:launch";
open(l); l;        // l is ready for writing but not for reading
↦ // type : MPtcp
↦ // mode : launch
↦ // name :
↦ // open : yes
↦ // read : not ready
↦ // write: ready

ring r; ideal i=x2+y,xyz+z,x2+y2;

write (l,quote(std(eval(i))));   // std(i) is computed on server
def j = read(l);j; // result of computation on server is read
↦ j[1]=z
↦ j[2]=y2-y
↦ j[3]=x2+y2

write(l, quote(getdump(mp_ll))); // server reads dump
dump(l);           //  dump is written to server (includes proc's)
read(l);           // result of previous write-command is read
killall("not", "link"); killall("proc"); // kills eveything, but links
↦ // ** killing the basering for level 0

write(l, quote(dump(mp_ll)));    // server writes dump
getdump(l);        // dump is read from server
read(l);           // result of previous write-command is read

close(l);          // server is shut down
listvar(all);      // same state as we had before "killall()"
↦ // mpsr_r0              [0]  ring
↦ // r                   [0]  *ring
↦ //      j                   [0]  ideal, 3 generator(s)
↦ //      i                   [0]  ideal, 3 generator(s)
```

```
↦ // l                    [0]  link

l = "MPtcp:";      // fork link declaration
open(l); l;        // Notice that name is "parent"
↦ // type : MPtcp
↦ // mode : fork
↦ // name : parent
↦ // open : yes
↦ // read : not ready
↦ // write: ready

write(l, quote(status(mp_ll, "name")));
read(l);           // and name of forked link is "child"
↦ child
write(l,quote(i)); // Child inherited vars and their values
read(l);
↦ _[1]=x2+y
↦ _[2]=xyz+z
↦ _[3]=x2+y2
close(l);          // shut down forked child
```

4.6.6 DBM links

DBM links provide access to data stored in a data base. Each entry in the data base consists of a (key_string, value_string) pair. Such a pair can be inserted with the command `write(link, key_string, value_string)`. By calling `write(link, key_string)`, the entry with key key_string is deleted from the data base. The value of an entry is returned by the command `read(link, key_string)`. With only one argument, `read(link)` returns the next key in the data base. Using this feature a data base can be scanned in order to access all entries of the data base.

If a data base with name `name` is opened for writing for the first time, two files (`name.pag` and `name.dir`) are automatically created, which contain the data base.

The DBM link describing string has to be one of the following:

1. `"DBM: " +` name
   opens the data base for reading (default mode).
2. `"DBM:r " +` name
   opens the data base for reading.
3. `"DBM:rw " +` name
   opens the data base for reading and writing.

Note, that `name` must be given without the suffix `.pag` ors `.dir`. The name may contain an (absolute) path.

Example:

```
link l="DBM:rw example";
```

```
write(l,"1","abc");
write(l,"3","XYZ");
write(l,"2","ABC");
l;
↦ // type : DBM
↦ // mode : rw
↦ // name : example
↦ // open : yes
↦ // read : ready
↦ // write: ready
close(l);
// read all keys (till empty string):
read(l);
↦ 1
read(l);
↦ 3
read(l);
↦ 2
read(l);
↦
// read data corresponding to key "1"
read(l,"1");
↦ abc
// read all data:
read(l,read(l));
↦ abc
read(l,read(l));
↦ XYZ
read(l,read(l));
↦ ABC
// close
close(l);
```

4.7 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. This applies also to the special list `#`. The expression `list()` is the empty list.

Note that a list stores the objects itself and not the names. Hence, if `L` is a list, `L[1]` for example has no name. A name, say `R`, can be created for `L[1]` by `def R=L[1];`. To store also the name of an object, say `r`, it can be added to the list with `nameof(r);`. Rings and qrings may be objects of a list.

Note: Unlike other assignments a ring as an element of a list is not a copy but another reference to the same ring.

4.7.1 list declarations

Syntax: `list` name = expression_list;
`list` name = list_expression;

Purpose: defines a list (of objects of possibly different types).

Default: empty list

Example:

```
list l=1,"str";
l[1];
↦ 1
l[2];
↦ str
ring r;
listvar(r);
↦ // r                      [0]  *ring
ideal i = x^2, y^2 + z^3;
l[3] = i;
l;
↦ [1]:
↦    1
↦ [2]:
↦    str
↦ [3]:
↦    _[1]=x2
↦    _[2]=z3+y2
listvar(r);     // the list l belongs now to the ring r
↦ // r                      [0]  *ring
↦ // l                      [0]  list, size: 3
↦ // i                      [0]  ideal, 2 generator(s)
```

4.7.2 list expressions

A list expression is:

1. the empty list `list()`
2. a list of expressions of any type
3. an identifier of type list
4. a function returning list
5. list expressions combined by the arithmetic operation `+`
6. a type cast to list

See Section 3.4.5 [Type conversion and casting], page 34.

Example:

```
list l = "hello",1;
l;
↦ [1]:
```

```
↦    hello
↦ [2]:
↦    1
l = list();
l;
↦ empty list
ring r =0,x,dp;
factorize((x+1)^2);
↦ [1]:
↦    _[1]=1
↦    _[2]=x+1
↦ [2]:
↦    1,2
list(1,2,3);
↦ [1]:
↦    1
↦ [2]:
↦    2
↦ [3]:
↦    3
```

4.7.3 list operations

`+` concatenation

`delete` deletes one element from list, returns new list

`insert` inserts or appends a new element to list, returns a new list

list_expression `[` int_expression `]`
is a list entry; the index 1 gives the first element.

Example:

```
list l1 = 1,"hello",list(-1,1);
list l2 = list(1,2,3);
l1 + l2;          // one new list
↦ [1]:
↦    1
↦ [2]:
↦    hello
↦ [3]:
↦    [1]:
↦       -1
↦    [2]:
↦       1
↦ [4]:
↦    1
↦ [5]:
↦    2
```

```
↦ [6]:
↦    3
list l3 =_;
l1,l2;            // two lists
↦ [1]:
↦    1
↦ [2]:
↦    hello
↦ [3]:
↦    [1]:
↦       -1
↦    [2]:
↦       1
↦ [1]:
↦    1
↦ [2]:
↦    2
↦ [3]:
↦    3
l2[2];
↦ 2
```

4.7.4 list related functions

`bareiss` returns a list of a matrix (lower triangular) and of an intvec (permutations of columns, see Section 5.1.2 [bareiss], page 103)

`betti` Betti numbers of a resolution (see Section 5.1.3 [betti], page 104)

`delete` deletes an element from a list (see Section 5.1.16 [delete], page 112)

`facstd` factorizing Groebner basis algorithm (see Section 5.1.26 [facstd], page 118)

`factorize` list of factors of a polynomial (see Section 5.1.27 [factorize], page 119)

`insert` inserts or appends a new element to a list (see Section 5.1.43 [insert], page 128)

`lres` minimal resolution (see Section 5.1.59 [lres], page 139)

`minres` minimize a free resolution (see Section 5.1.64 [minres], page 142)

`mres` minimal free resolution of an ideal resp. module, also minimizing the first module (see Section 5.1.67 [mres], page 144)

`names` list of all userdefined variable names (see Section 5.1.71 [names], page 146)

`res` minimal free resolution of an ideal resp. module (see Section 5.1.96 [res], page 162)

`size` number of entries (see Section 5.1.102 [size], page 166)

`sres` free resolution of an ideal resp. module given by a standard base (see Section 5.1.104 [sres], page 167)

4.8 map

Maps are ring maps from a preimage ring into the basering. Note that the target of a map is ALWAYS the actual basering. Maps between rings with different coefficient fields are possible, coefficients are mapped in the following way:

- $Z/p \mapsto Q \qquad [i]_p \to i \in [-p/2,\, p/2] \subseteq Z$
- $Z/p \mapsto Z/p' \qquad [i]_p \in Z/p \to i \in [-p/2,\, p/2] \subseteq Z,\ i \to [i]_{p'} \in Z/p'$

and as usual:

- $Q \mapsto Z/p$
- $Q \mapsto (Z/p)(a,\ldots)$
- $Q \mapsto Q(a,\ldots)$
- $Z/p \mapsto (Z/p)(a,\ldots)$

See Section 5.1.41 [imap], page 127; Section 5.1.28 [fetch], page 119; Section 5.1.109 [subst], page 171.

4.8.1 map declarations

Syntax: `map` name `=` preimage_ring_name `,` ideal_expression `;`

Purpose: defines a ring map from preimage_ring to basering.
Maps the variables of the preimage ring to the generators of the ideal. If the ideal contains less elements than variables in the preimage_ring the remaining variables are mapped to 0, if the ideal contains more elements these are ignored. The image ring is always the actual basering. For the mapping of coefficients from different fields see Section 4.8 [map], page 78.

Default: none

Note: There are standard mappings for maps which are close to the identity map: `fetch` and `imap`.

The name of a map serves as the function which maps objects from the preimage_ring into the basering. These objects must be defined by names (no evaluation in the preimage ring is possible).

Example:

```
ring r1=32003,(x,y,z),dp;
ideal i=x,y,z;
ring r2=32003,(a,b),dp;
map f=r1,a,b,a+b;
// maps from r1 to r2,
// x -> a
// y -> b
// z -> a+b
f(i);
↦ _[1]=a
↦ _[2]=b
```

```
↦ _[3]=a+b
// operations like f(i[1]) or f(i*i) are not allowed
ideal i=f(i);
// objects in different rings may have the same name
map g   = r2,a2,b2;
map phi = g(f);
// composition of map f and g
// maps from r1 to r2,
// x -> a2
// y -> b2
// z -> a2+b2
phi(i);
↦ _[1]=a2
↦ _[2]=b2
↦ _[3]=a2+b2
```

See Section 4.8 [map], page 78; Section 4.2.2 [ideal expressions], page 50; Section 4.16 [ring], page 95; Section 5.1.41 [imap], page 127; Section 5.1.28 [fetch], page 119.

4.8.2 map expressions

A map expression is:

1. an identifier of type map
2. a function returning map
3. map expressions combined by composition using parentheses (`(`, `)`)

4.8.3 map operations

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

map_expression `[` int_expressions `]`
is a map entry (the image of the corresponding variable)

Example:

```
ring r=0,(x,y),dp;
map f=r,y,x;    // the map f permutes the variables
f;
↦ f[1]=y
↦ f[2]=x
poly p=x+2y3;
f(p);
↦ 2x3+y
map g=f(f);    // the map g defined as  f^2, the involtution of f
g;
```

```
↦ g[1]=x
↦ g[2]=y
g(p) == p;
↦ 1
```

4.9 matrix

Objects of type matrix are matrices with polynomial entries. Like polynomials they can only be defined or accessed with respect to a basering. In order to compute with matrices having integer or rational entries define a ring with characteristic 0 and at least one variable.

A matrix can be multiplied by and added to a poly; in this case the poly is converted into a matrix of the right size with the poly on the diagonal.

If A is a matrix then the assignment `module M=A;` or `module M=module(A);` creates a module generated by the columns of A. Note that the trailing zero columns of A may be deleted by module operations with M.

4.9.1 matrix declarations

Syntax: `matrix` name[rows][cols] = list_of_poly_expressions ;
`matrix` name = matrix_expression ;

Purpose: defines a matrix (of polynomials).
The given poly_list fills up the matrix beginning with the first row from the left to the right, then the second row and so on. If the poly_list contains less than rows*cols elements, the matrix is filled up with zeros; if it contains more elements, then only the first rows*cols elements are used. If the right-hand side is a matrix expression the matrix on the left-hand side becomes the same size as the right-hand side, otherwise the size is determined by the left-hand side. If the size is omitted a 1x1 matrix is created.

Default: 0 (1 x 1 matrix)

Example:

```
int ro = 3;
ring r = 32003,(x,y,z),dp;
poly f=xyz;
poly g=z*f;
ideal i=f,g,g^2;
matrix m[ro][3] = x3y4, 0, i, f ; // a 3 x 3 matrix
m;
↦ m[1,1]=x3y4
↦ m[1,2]=0
↦ m[1,3]=xyz
↦ m[2,1]=xyz2
↦ m[2,2]=x2y2z4
↦ m[2,3]=xyz
↦ m[3,1]=0
↦ m[3,2]=0
```

```
↦ m[3,3]=0
print(m);
↦ x3y4,0,     xyz,
↦ xyz2,x2y2z4,xyz,
↦ 0,   0,     0
matrix A;   // the 1 x 1 zero matrix
matrix B[2][2] = m[1..2, 2..3]; //defines a submatrix
print(B);
↦ 0,     xyz,
↦ x2y2z4,xyz
matrix C=m; // defines C as a 3 x 3 matrix equal to m
print(C);
↦ x3y4,0,     xyz,
↦ xyz2,x2y2z4,xyz,
↦ 0,   0,     0
```

4.9.2 matrix expressions

A matrix expression is:

1. an expression list of poly expressions
2. an identifier of type matrix
3. a function returning matrix
4. matrix expressions combined by the arithmetic operations `+`, `-` or `*`
5. a type cast to matrix

Example:

```
ring r=0,(x,y),dp;;
poly f= x3y2 + 2x2y2 +2;
matrix H = jacob(jacob(f));    // the Hessian of f
matrix mc = coef(f,y);
print(mc);
↦ y2,    1,
↦ x3+2x2,2
module MD = [x+y,1,x],[x+y,0,y];
matrix M = MD;
print(M);
↦ x+y,x+y,
↦ 1,  0,
↦ x,  y
```

4.9.3 matrix operations

`+`	addition with matrix or poly; the poly is converted into a diagonal matrix
`-`	negation or subtraction with matrix or poly; the poly is converted into a diagonal matrix

`*` multiplication with matrix or poly; the poly is converted into a diagonal matrix

`==`, `<>`, `!=` comparison

matrix_expression `[` int_expression`,` int_expression `]`
is a matrix entry, where the first index indicates the row and the second the column

Example:

```
ring r=32003,x,dp;
matrix A[3][3] = 1,3,2,5,0,3,2,4,5;
print(A);
↦ 1,3,2,
↦ 5,0,3,
↦ 2,4,5
matrix E[3][3]; E = E + 1;  // the unit matrix
matrix B =x*E - A;
print(B);
↦ x-1,-3,-2,
↦ -5, x, -3,
↦ -2, -4,x-5
det(B);        // the characteristic polynomial of A
↦ x3-6x2-26x+29
A*A*A - 6 * A*A - 26*A == -29*E;  // Cayley-Hamilton
↦ 1
vector v =[ x,-1,x2];
A*v;
↦ _[1,1]=2x2+x-3
↦ _[2,1]=3x2+5x
↦ _[3,1]=5x2+2x-4
matrix m[2][2]=1,2,3;
print(m-transpose(m));
↦ 0,-1,
↦ 1,0
```

4.9.4 matrix related functions

`bareiss` Gauss-Bareiss algorithm (see Section 5.1.2 [bareiss], page 103)

`coef` matrix of coefficients and monomials (see Section 5.1.9 [coef], page 107)

`coeffs` matrix of coefficients (see Section 5.1.10 [coeffs], page 108)

`det` determinant (see Section 5.1.17 [det], page 113)

`diff` partial derivative (see Section 5.1.18 [diff], page 113)

`jacob` Jacobi matrix (see Section 5.1.46 [jacob], page 130)

`koszul` Koszul matrix (see Section 5.1.51 [koszul], page 134)

`lift` lift-matrix (see Section 5.1.56 [lift], page 137)

`liftstd` standard basis and transformation matrix computation (see Section 5.1.57 [liftstd], page 137)

`minor` set of minors of a matrix (see Section 5.1.63 [minor], page 141)

`ncols` number of columns (see Section 5.1.72 [ncols], page 147)

`nrows` number of rows (see Section 5.1.75 [nrows], page 149)

`print` nice print format (see Section 5.1.87 [print], page 156)

`size` number of matrix entries (see Section 5.1.102 [size], page 166)

`subst` substitute a ring variable (see Section 5.1.109 [subst], page 171)

`trace` trace of a matrix (see Section 5.1.112 [trace], page 174)

`transpose`
transpose a matrix (see Section 5.1.113 [transpose], page 174)

`wedge` wedge product (see Section 5.1.119 [wedge], page 177)

4.10 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. The assignments `module M=v1,...,vk;` `matrix A=M;` creates the presentation matrix of size n x k for R^n/M, i.e., the columns of A are the vectors $v_1, \ldots, v_k$ which generate M (cf. Section B.1 [Representation of mathematical objects], page 239).

4.10.1 module declarations

Syntax: `module` name = list_of_vector_expressions ;
`module` name = module_expression ;

Purpose: defines a module.

Default: [0]

Example:

```
ring r=0,(x,y,z),(c,dp);
vector s1 = [x2,y3,z];
vector s2 = [xy,1,0];
vector s3 = [0,x2-y2,z];
poly   f  = xyz;
module m = s1, s2-s1,f*(s3-s1);
m;
↦ m[1]=[x2,y3,z]
↦ m[2]=[-x2+xy,-y3+1,-z]
↦ m[3]=[-x3yz,-xy4z+x3yz-xy3z]
// show m in matrix format (columns generate m)
print(m);
↦ x2,-x2+xy,-x3yz,
↦ y3,-y3+1, -xy4z+x3yz-xy3z,
↦ z, -z,    0
```

4.10.2 module expressions

A module expression is:

1. an expression list of vector expressions
2. an identifier of type module
3. a function returning module
4. module expressions combined by the arithmetic operation `+`
5. multiplication of a module expressions with an ideal or poly expression: `*`
6. a type cast to module

See Section 4.2 [ideal], page 50; Section 4.12 [poly], page 89; Section 3.4.5 [Type conversion and casting], page 34; Section 4.18 [vector], page 99.

4.10.3 module operations

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

`*` multiplication with ideal or poly, but not 'module' * 'module'

module_expression `[` int_expression `,` int_expression `]`
is a module entry, where the first index indicates the row and the second the column

module_expressions `[` int_expression `]`
is a vector, where the index indicates the column

Example:

```
ring r=0,(x,y,z),dp;
module m=[x,y],[0,0,z];
print(m*(x+y));
↦ x2+xy,0,
↦ xy+y2,0,
↦ 0,    xz+yz
```

4.10.4 module related functions

`coeffs` matrix of coefficients (see Section 5.1.10 [coeffs], page 108)

`degree` multiplicity, dimension and codimension of the module of leading terms (see Section 5.1.15 [degree], page 112)

`diff` partial derivative (see Section 5.1.18 [diff], page 113)

`dim` Krull dimension of free module over the basering modulo the module of leading terms (see Section 5.1.19 [dim], page 114)

`eliminate`
elimination of variables (see Section 5.1.21 [eliminate], page 115)

`freemodule`
: the free module of given rank (see Section 5.1.33 [freemodule], page 122)

`groebner` Groebner basis computation (a wrapper around `std,stdhilb,stdfglm`,...) (see Section 5.1.37 [groebner], page 124)

`hilb` Hilbert function of a standard basis (see Section 5.1.39 [hilb], page 125)

`homog` homogenization with respect to a variable (see Section 5.1.40 [homog], page 126)

`interred` interreduction of a module (see Section 5.1.44 [interred], page 129)

`intersect`
: module intersection (see Section 5.1.45 [intersect], page 130)

`jet` Taylor series up to a given order (see Section 5.1.47 [jet], page 131)

`kbase` vector space basis of free module over the basering modulo the module of leading terms (see Section 5.1.48 [kbase], page 132)

`lead` initial module (see Section 5.1.52 [lead], page 134)

`lift` lift-matrix (see Section 5.1.56 [lift], page 137)

`liftstd` standard basis and transformation matrix computation (see Section 5.1.57 [liftstd], page 137)

`lres` minimal resolution (see Section 5.1.59 [lres], page 139)

`minbase` minimal generating set of a homogeneous ideal resp. module or an ideal resp. module over a local ring

`modulo` represents $(h1 + h2)/h1 = h2/(h1 \cap h2)$ (see Section 5.1.65 [modulo], page 143)

`mres` minimal free resolution of an ideal resp. module, also minimizing the given module (see Section 5.1.67 [mres], page 144)

`mult` multiplicity resp. degree of the module of leading terms (see Section 5.1.69 [mult], page 145)

`ncols` number of columns (see Section 5.1.72 [ncols], page 147)

`nrows` number of rows (see Section 5.1.75 [nrows], page 149)

`print` nice print format (see Section 5.1.87 [print], page 156)

`prune` minimize the embedding into a free module (see Section 5.1.88 [prune], page 157)

`qhweight` quasihomogeneous weights of an ideal resp. module (see Section 5.1.89 [qhweight], page 157)

`quotient` ideal quotient (see Section 5.1.91 [quotient], page 158)

`reduce` normalform with respect to a standard basis (see Section 5.1.94 [reduce], page 161)

`res` minimal free resolution of an ideal resp. module but not changing the given ideal resp. module (see Section 5.1.96 [res], page 162)

`simplify` simplify a set of vectors (see Section 5.1.101 [simplify], page 165)

`size` number of non-zero generators (see Section 5.1.102 [size], page 166)

`sortvec` permutation for sorting ideals/modules (see Section 5.1.103 [sortvec], page 167)

`sres` free resolution of a standard basis (see Section 5.1.104 [sres], page 167)

`std` standard basis computation (see Section 5.1.106 [std], page 169, Section 5.1.57 [liftstd], page 137)

`subst` substitute a ring variable (see Section 5.1.109 [subst], page 171)

`syz` computation of the first syzygy module (see Section 5.1.111 [syz], page 173)

`vdim` vector space dimension of free module over the basering modulo module of leading terms (see Section 5.1.118 [vdim], page 176)

`weight` "optimal" weights (see Section 5.1.120 [weight], page 177)

4.11 number

Numbers are elements from the coefficient field. They can only be defined or accessed with respect to a basering which determines the coefficient field.

4.11.1 number declarations

Syntax: `number` name = number_expression ;

Purpose: defines a number.

Default: 0

Example:

```
ring r = 32003,(x,y,z),dp;
number n = 4/6;
n;
↦ -10667
ring r0 = 0,x,dp;
number n = 4/6;
n;
↦ 2/3
ring R=real,x,dp;
number n=4/6;
n;
↦ 6.667e-01
n=0.25e+2;
n;
↦ 2.500e+01
```

4.11.2 number expressions

A number expression is:

1. a rational number (there are NO spaces allowed inside a rational number, see Section 4.3.2 [int expressions], page 55)
2. a floating point number (if the coefficient field is `real`):
   <digits>.<digits>e<sign><digits>
3. an identifier of type number
4. a function returning number
5. an int expression (see Section 3.4.5 [Type conversion and casting], page 34)

6. number expressions combined by the arithmetic operations +, -, *, /, ^, or **.
7. a type cast to number

Example:

```
// the following expressions are in any ring int expressions
2 / 3;
↦ 0
4/ 8;
↦ 0
2 /2;   // the notation of / for div might change in the future
↦ 1
ring r0=0,x,dp;
2/3, 4/8, 2/2 ; // are numbers
↦ 2/3 1/2 1

poly f = 2x2 +1;
leadcoef(f);
↦ 2
typeof(_);
↦ number
ring rr =real,x,dp;
1.7e-2; 1.7e+2; // are valid (but  1.7e2 not), if the field is ‘real‘
↦ 1.700e-02
↦ 1.700e+02
ring rp = (31,t),x,dp;
2/3, 4/8, 2/2 ; // are numbers
↦ (11) (-15) (1)
poly g = (3t2 +1)*x2 +1;
leadcoef(g);
↦ (3t2+1)
typeof(_);
↦ number
par(1);
↦ (t)
typeof(_);
↦ number
```

See Section 4.16 [ring], page 95; Section 3.4.5 [Type conversion and casting], page 34.

4.11.3 number operations

+	addition
-	negation or subtraction
*	multiplication
/	division
^, **	power, exponentiation (by an integer)

`<=, >=, ==, <>`
: comparison

Note: quotient and exponentiation is only recognized as a number expression is if is already a number, see Section 6.3 [Miscellaneous oddities], page 193.
To acces the denominator or nominator of a rtional number see Chapter 6 [Tricks and pitfalls], page 190.
For the behavior of comparison operators in rings with basering different from real or the rational numbers, see Section 4.3.5 [boolean expressions], page 58.

Example:

```
ring r=0,x,dp;
number n = 1/2 +1/3;
n;
↦ 5/6
n/2;
↦ 5/12
1/2/3;
↦ 1/6
1/2 * 1/3;
↦ 1/6
n = 2;
n^-2;
↦ 1/4
// the following oddities appear here
2/(2+3);
↦ 0
number(2)/(2+3);
↦ 2/5
2^-2;
↦    ? exponent must be non-negative
↦    ? error occurred in Z line 12: `2^-2;`
number(2)^-2;
↦ 1/4
3/4>=2/5;
↦ 1
2/6==1/3;
↦ 1
```

4.11.4 number related functions

`cleardenom`
: cancels denominators of numbers in poly and divide by its content (see Section 5.1.7 [cleardenom], page 107)

`leadcoef` coefficient of the leading term (see Section 5.1.53 [leadcoef], page 135)

`par` n-th parameter of the basering (see Section 5.1.81 [par], page 154)

`pardeg` degree of a number in ring parameters (see Section 5.1.82 [pardeg], page 154)

`parstr` string form of ring parameter (see Section 5.1.83 [parstr], page 155)

4.12 poly

Polynomials are the basic data for all main algorithms in SINGULAR. They consist of finitely many terms (coefficient*power product) which are combined by the usual polynomial operations (see Section 4.12.2 [poly expressions], page 89). 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;
```

4.12.1 poly declarations

Syntax: `poly` name `=` poly_expression `;`

Purpose: defines a polynomial.

Default: 0

Example:

```
ring r = 32003,(x,y,z),dp;
poly s1  = x3y2+151x5y+186xy6+169y9;
poly s2  = 1*x^2*y^2*z^2+3z8;
poly s3  = 5/4x4y2+4/5*x*y^5+2x2y2z3+y7+11x10;
int a,b,c,t=37,5,4,1;
poly f=3*x^a+x*y^(b+c)+t*x^a*y^b*z^c;
```

4.12.2 poly expressions

A poly expression is (optional parts in square brackets):

1. a monomial (there are NO spaces allowed inside a monomial)

   `[coefficient] ring_variable [ exponent] [ring_variable [exponent] ...]`

   monomials which contain an indexed ring variable must be built from `ring_variable` and `coefficient` with the operations `*` and `^`
2. an identifier of type poly
3. a function returning poly
4. poly expressions combined by the arithmetic operations `+`, `-`, `*`, `/`, or `^`
5. a type cast to poly

Example:

```
2x, x3, 2x2y3, xyz, 2xy2; //  are monomials
2*x, x^3, 2*x^2*y^3, x*y*z, 2*x*y^2; // are poly expressions
```

```
2*x(1); // is a valid poly expression, but not 2x(1) (a syntax error)
2*x^3;  // is a valid poly expression equal to 2x3 (a valid monomial)
        // but not equal to 2x^3 which will be interpreted as (2x)^3
        // since 2x is a monomial
ring r=0,(x,y),dp;
poly f = 10x2y3 +2x2y2-2xy+y -x+2;
lead(f);
↦ 10x2y3
simplify(f,1);     // normalize leading coefficient
↦ x2y3+1/5x2y2-1/5xy-1/10x+1/10y+1/5
poly g = 1/2x2 + 1/3y;
cleardenom(g);
↦ 3x2+2y
int i = 102;
poly(i);
↦ 102
typeof(_);
↦ poly
```

See Section 4.16 [ring], page 95; Section 3.4.5 [Type conversion and casting], page 34.

4.12.3 poly operations

`+` addition

`-` negation or subtraction

`*` multiplication

`/` division by a monomial, non divisible terms yield 0

`^`, `**` power by an integer

`<`, `<=`, `>`, `>=`, `==`, `<>`
comparison (of leading terms w.r.t. monomial ordering)

poly_expression `[` intvec_expression `]`
the monomial at the indicated place w.r.t. the monomial order

Example:

```
ring R=0,(x,y),dp;
poly f = x3y2 + 2x2y2 + xy - x + y + 1;
f;
↦ x3y2+2x2y2+xy-x+y+1
f + x5 + 2;
↦ x5+x3y2+2x2y2+xy-x+y+3
f * x2;
↦ x5y2+2x4y2+x3y-x3+x2y+x2
(x+y)/x;
↦ 1
f/3x2;
```

```
↦ 1/3xy2+2/3y2
x5 > f;
↦ 1
x<=y;
↦ 0
x>y;
↦ 1
ring r=0,(x,y),ds;
poly f = fetch(R,f);
f;
↦ 1-x+y+xy+2x2y2+x3y2
x5 > f;
↦ 0
f[2..4];
↦ -x+y+xy
size(f);
↦ 6
f[size(f)+1]; f[-1];    // monoms out of range are 0
↦ 0
↦ 0
intvec v = 6,1,3;
f[v];          // the polynom built from the 1st, 3rd and 6th monom of f
↦ 1+y+x3y2
```

4.12.4 poly related functions

`cleardenom`
: cancels denominators of numbers in poly and divide by its content (see Section 5.1.7 [cleardenom], page 107)

`coef`
: matrix of coefficients and monomials (see Section 5.1.9 [coef], page 107)

`coeffs`
: matrix of coefficients (see Section 5.1.10 [coeffs], page 108)

`deg`
: degree (see Section 5.1.14 [deg], page 111)

`det`
: determinant (see Section 5.1.17 [det], page 113)

`diff`
: partial derivative (see Section 5.1.18 [diff], page 113)

`extgcd`
: Bezout representation of gcd (see Section 5.1.25 [extgcd], page 117)

`factorize`
: factorize polynomial (see Section 5.1.27 [factorize], page 119)

`finduni`
: find univariate polynomials in a zero dimensional ideal (see Section 5.1.32 [finduni], page 122)

`gcd`
: greatest common divisor (see Section 5.1.34 [gcd], page 123)

`homog`
: homogenization (see Section 5.1.40 [homog], page 126)

`jacob`
: ideal resp. matrix of all partial derivatives (see Section 5.1.46 [jacob], page 130)

`lead`
: leading monomial (see Section 5.1.52 [lead], page 134)

`leadcoef`
: coefficient of the leading term (see Section 5.1.53 [leadcoef], page 135)

`leadexp`
: the exponent vector of the leading monomial (see Section 5.1.54 [leadexp], page 135)

`jet`	monomials with degree smaller k+1 (see Section 5.1.47 [jet], page 131)
`ord`	degree of the leading monomial (see Section 5.1.79 [ord], page 153)
`qhweight`	quasihomogenous weights (see Section 5.1.89 [qhweight], page 157)
`reduce`	normal form with respect to a standard base (see Section 5.1.94 [reduce], page 161)
`rvar`	test for ring variable (see Section 5.1.99 [rvar], page 164)
`simplify`	normalize a polynomial (see Section 5.1.101 [simplify], page 165)
`size`	number of monomials (see Section 5.1.102 [size], page 166)
`subst`	substitute a ring variable (see Section 5.1.109 [subst], page 171)
`trace`	trace of a matrix (see Section 5.1.112 [trace], page 174)
`var`	the indicated variable of the ring (see Section 5.1.116 [var], page 176)
`varstr`	variable in string form (see Section 5.1.117 [varstr], page 176)

4.13 proc

Procedures are sequences of SINGULAR commands in a special format. They are used to extend the set of SINGULAR commands with user defined commands. Once a procedure is defined it can be used as any other SINGULAR command. Procedures may be defined by either typing them on the command line or by loading them from a file. For a detailed description on the concept of procedures in SINGULAR see Section 3.6 [Procedures], page 40. A file containing procedure definitions which comply with certain syntax rules is called a library. Such a file is loaded using the command `LIB`. For more information on libraries see Section 3.7 [Libraries], page 44.

4.13.1 proc declaration

Syntax:

```
[static] proc proc_name [parameter_list]
["help_text"]
{
  procedure_body
}
[example
{
  sequence_of_commands;
}]
```

Purpose: defines a new function, the `proc` proc_name, with the additional information help_text, which is copied to the screen by `help proc_name;` and the `example` section which is executed by `example proc_name;`.
The help_text, the parameter_list, and the example section are optional. The default for a parameter_list is `(list #)`, see Section 3.6.3 [Parameter list], page 42. The help and example sections are ignored if the procedure is defined interactively, i.e., if it was not loaded from a file by a Section 5.1.55 [LIB], page 136 command.
Specifying `static` in front of the proc-definition (in a library file) makes this procedure local to the library, i.e., accessible only for the other procedures in the same library, but not for the users.

Example:

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

See Section 5.1.55 [LIB], page 136; Section 3.7 [Libraries], page 44; Section 3.6 [Procedures], page 40

4.14 qring

SINGULAR offers the opportunity to calculate in quotient rings (factor rings), i.e., rings modulo an ideal. The ideal has to be given as a standard basis. For a detailed description of the concept of rings and quotient rings see Section 3.2 [Rings and orderings], page 20.

4.14.1 qring declaration

Syntax: `qring` name = ideal_expression ;

Default: none

Purpose: declares a quotient ring as the basering modulo ideal_expression. Sets it as current basering.

Example:

```
ring r=0,(x,y,z),dp;
ideal i=xy;
qring q=std(i);
basering;
↦ //   characteristic : 0
↦ //   number of vars : 3
↦ //        block   1 : ordering dp
↦ //                  : names    x y z
↦ //        block   2 : ordering C
↦ // quotient ring from ideal
↦ _[1]=xy
```

4.15 resolution

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

Note: to access the elements of a resolution, it has to be assigned to a list, which also completes computations and may therefore take time, (resp. an access directly with the brackets `[ , ]` causes implicitly a cast to a list).

4.15.1 resolution declarations

Syntax: `resolution` name `=` resolution_expression `;`

Purpose: defines a resolution.

Default: none

Example:

```
ring R;
ideal i=z2,x;
resolution re=res(i,0);
re;
↦  1      2      1      0
↦ R  <-- R  <-- R  <-- R
↦
↦ 0      1      2      3
↦ resolution not minimized yet
↦
betti(re);
↦ 1,1,0,
↦ 0,1,1
list l = re;
l;
↦ [1]:
↦    _[1]=x
↦    _[2]=z2
↦ [2]:
↦    _[1]=-z2*gen(1)+x*gen(2)
↦ [3]:
↦    _[1]=0
```

4.15.2 resolution expressions

A resolution expression is:

1. an identifier of type resolution
2. a function returning a resolution
3. a type cast to resolution from a list of ideals resp. modules..

See Section 3.4.5 [Type conversion and casting], page 34.

4.15.3 resolution related functions

`betti`	Betti numbers of a resolution (see Section 5.1.3 [betti], page 104)
`lres`	minimal resolution (see Section 5.1.59 [lres], page 139)
`minres`	minimize a free resolution (see Section 5.1.64 [minres], page 142)
`mres`	minimal free resolution of an ideal resp. module, also minimizing the given ideal resp. module (see Section 5.1.67 [mres], page 144)
`res`	minimal free resolution of an ideal resp. module but not changing the given ideal resp. module (see Section 5.1.96 [res], page 162)
`sres`	free resolution of a standard basis (see Section 5.1.104 [sres], page 167)

4.16 ring

Rings are used to describe properties of polynomials, ideals etc. Almost all computations in SINGULAR require a basering. For a detailed description of the concept of rings see Section 3.2 [Rings and orderings], page 20.

4.16.1 ring declarations

Syntax: `ring` name `=` coefficient_field`,` `(` names_of_ring_variables `),` `(` ordering `);`

Default: `32003,(x,y,z),(dp,C);`

Purpose: declares a ring and sets it as the actual basering.

The coefficient_field is given by one of the following:

1. a non-negative int_expression less or equal 32003.
2. an expression_list of an int_expression and one or more names.
3. the name `real`.

The names_of_ring_variables is a list of names or indexed names.

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

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

For the definition of the orderings, see Section 3.2.3 [Term orderings], page 24, Section B.2 [Monomial orderings], page 240.

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

4.16.2 ring related functions

`charstr`	description of the coeffinet field of a ring (see Section 5.1.6 [charstr], page 106)
`keepring`	move ring to next upper level (see Section 5.2.7 [keepring], page 182)
`npars`	number of ring parameters (see Section 5.1.73 [npars], page 148)
`nvars`	number of ring variables (see Section 5.1.76 [nvars], page 150)
`ordstr`	monomial ordering of a ring (see Section 5.1.80 [ordstr], page 154)
`parstr`	names of all ring parameters or the name of the n-th ring parameter (see Section 5.1.83 [parstr], page 155)
`qring`	quotient ring (see Section 4.14 [qring], page 93)
`setring`	set a new basering (see Section 5.1.100 [setring], page 164)
`varstr`	names of all ring variables or the name of the n-th ring variable (see Section 5.1.117 [varstr], page 176)

4.17 string

Variables of type `string` are used for output (almost every type can be "converted" to `string`) and for creating new commands at runtime (see Section 5.1.23 [execute], page 117). They are also return values of certain interpreter related functions (see Section 5.1 [Functions], page 102). String constants consist of a sequence of ANY characters (including newline!) between a starting " and a closing ". There is also a string constant `newline`, which is the newline character. The `+` sign "adds" strings, "" is the empty string (hence strings form a semigroup). Strings may be used to comment the output of a computation or to give it a nice format. Strings may also be used for intermediate conversion of one type into another.

```
string s="Hi";
string s1="a string with new line at the end"+newline;
string s2="another string with new line at the end
";
s;s1;s2;
↦ Hi
↦ a string with new line at the end
↦
↦ another string with new line at the end
↦
ring r; ideal i=std(ideal(x,y^3));
"dimension of i =",dim(i),", multiplicity of i =",mult(i);
↦ dimension of i = 1 , multiplicity of i = 3
"dimension of i = "+string(dim(i))+", multiplicity of i = "+string(mult(i));
↦ dimension of i = 1, multiplicity of i = 3
"a"+"b","c";
↦ ab c
```

A comma between two strings makes an expression list out of them (such a list is printed with as a separating blank between), while a + concatenates strings.

4.17.1 string declarations

Syntax: `string` name = string_expression ;

Purpose: defines a string variable.

Default: "" (the empty string)

Example:

```
string s1="Now I know";
string s2="how to encode a \" in a string...";
string s=s1+" "+s2; // concatenation of 3 strings
s;
↦ Now I know how to encode a " in a string...
s1,s2;   // 2 strings, separated by a blank in the output:
↦ Now I know how to encode a " in a string...
```

4.17.2 string expressions

A string expression is:

1. a sequence of characters between two unescaped quotes (")
2. a list of string expressions
3. an identifier of type string
4. a function returning string
5. a substring (using the bracket operator)
6. a type cast to string
7. string expressions combined by the operation +.

Example:

```
// string_expression[start, length] : a substring
// (possibly filled up with blanks)
// the substring of s starting at position 2
// with a length of 4
string s="123456";
s[2,4];
↦ 2345
"abcd"[2,2];
↦ bc
// string_expression[position] : a character from a string
s[3];
↦ 3
// string_expression[position..position] :
// a substring starting at the first position up to the second
```

```
// given position
s[2..4];
↦ 2 3 4
// a function returning a string
typeof(s);
↦ string
```

See Section 3.4.5 [Type conversion and casting], page 34.

4.17.3 string operations

`+` concatenation

`<=`, `>=`, `==`, `<>`
comparison (lexicographic with respect to the ASCII encoding)

string_expression `[` int_expression `]`
is a character of the string; the index 1 gives the first character.

string_expression `[` int_expression`,` int_expression `]`
is a substring, where the first argument is the start index and the second is the length of the substring, filled up with blanks if the length exceeds the total size of the string

string_expression `[` intvec_expression `]`
is a expression list of characters from the string

Example:

```
string s="abcde";
s[2];
↦ b
s[3,2];
↦ cd
">>"+s[1,10]+"<<";
↦ >>abcde     <<
s[2]="BC"; s;
↦ aBcde
intvec v=1,3,5;
s=s[v]; s;
↦ ace
s="123456"; s=s[3..5]; s;
↦ 345
```

4.17.4 string related functions

`charstr` description of the coeffinet field of a ring (see Section 5.1.6 [charstr], page 106)

`execute` executing string as command (see Section 5.1.23 [execute], page 117)

`find` position of a substring in a string (see Section 5.1.31 [find], page 121)

`names` list of strings of all userdefined variable names (see Section 5.1.71 [names], page 146)

`nameof` name of an object (see Section 5.1.70 [nameof], page 146)

`option` lists all defined options (see Section 5.1.78 [option], page 150)

`ordstr` monomial ordering of a ring (see Section 5.1.80 [ordstr], page 154)

`parstr` names of all ring parameters or the name of the n-th ring parameter (see Section 5.1.83 [parstr], page 155)

`read` read a file (see Section 5.1.93 [read], page 160)

`size` length of a string (see Section 5.1.102 [size], page 166)

`typeof` type of an object (see Section 5.1.115 [typeof], page 175)

`varstr` names of all ring variables or the name of the n-th ring variable (see Section 5.1.117 [varstr], page 176)

4.18 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 biggest index of a generator with nonzero 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. Like polynomials they can only be defined or accessed with respect to this basering. (E.g., if `f` and `g` are polynomials then `f*gen(1)+g*gen(3)+gen(4)` may also be written as `[f,0,g,1]` or as `[f,0,g,1,0]`.). Note that the elements of a vector have to be surrounded by square brackets (`[` , `]`). (cf. Section B.1 [Representation of mathematical objects], page 239)

4.18.1 vector declarations

Syntax: `vector` name `=` vector_expression `;`

Purpose: defines a vector of polynomials (an element of a free module).

Default: [0]

Example:

```
ring r=0,(x,y,z),(c,dp);
poly s1 = x2;
poly s2 = y3;
poly s3 = z;
vector v = [s1, s2-s1, s3-s1]+ s1*gen(5);
// v is a vector in the free module of rank 5
v;
↦ [x2,y3-x2,-x2+z,0,x2]
```

4.18.2 vector expressions

A vector expression is:

1. an identifier of type vector

2. a function returning vector
3. a poly expression (via the canonical embedding `p` $\mapsto$ `p*gen(1)`)
4. vector expressions combined by the arithmetic operations `+` or `-`
5. a poly expression and a vector expression combined by the arithmetic operation `*`
6. a type cast to vector using the brackets `[ , ]`

Example:

```
// ordering gives priority to components:
ring rr=0,(x,y,z),(c,dp);
vector v=[x2+y3,2,0,x*y]+gen(6)*x6;
v;
↦ [y3+x2,2,0,xy,0,x6]
vector w=[z3-x,3y];
v-w;
↦ [y3-z3+x2+x,-3y+2,0,xy,0,x6]
v*(z+x);
↦ [xy3+y3z+x3+x2z,2x+2z,0,x2y+xyz,0,x7+x6z]
```

See Section 4.16 [ring], page 95; Section 3.4.5 [Type conversion and casting], page 34.

4.18.3 vector operations

`+` addition

`-` negation or subtraction

`/` division by a monomial, not divisible terms yield 0

`<`, `<=`, `>`, `>=`, `==`, `<>`
comparison of leading terms w.r.t. monomial ordering

vector_expression `[` int_expressions `]`
is a vector entry; the index 1 gives the first entry.

Example:

```
ring R=0,(x,y),(c,dp);
[x,y]-[1,x];
↦ [x-1,-x+y]
[1,2,x,4][3];
↦ x
```

4.18.4 vector related functions

`cleardenom`
quotient of a vector by its content (see Section 5.1.7 [cleardenom], page 107)

`coeffs`	matrix of coefficients (see Section 5.1.10 [coeffs], page 108)
`deg`	degree (see Section 5.1.14 [deg], page 111)
`diff`	partial derivative (see Section 5.1.18 [diff], page 113)
`gen`	i-th generator (see Section 5.1.35 [gen], page 123)
`homog`	homogenization (see Section 5.1.40 [homog], page 126)
`jet`	k-jet: monomials with degree smaller k+1 (see Section 5.1.47 [jet], page 131)
`lead`	leading monomial (see Section 5.1.52 [lead], page 134)
`leadcoef`	leading coefficient (see Section 5.1.53 [leadcoef], page 135)
`leadexp`	the exponent vector of the leading monomial (see Section 5.1.54 [leadexp], page 135)
`nrows`	number of rows (see Section 5.1.75 [nrows], page 149)
`ord`	degree of the leading monomial (see Section 5.1.79 [ord], page 153)
`reduce`	normal form with respect to a standard base (see Section 5.1.94 [reduce], page 161)
`simplify`	normalize a vector (see Section 5.1.101 [simplify], page 165)
`size`	number of monomials (see Section 5.1.102 [size], page 166)
`subst`	substitute a ring variable (see Section 5.1.109 [subst], page 171)

5 Functions and system variables

5.1 Functions

The general syntax of a function is
[target =] function_name (<arguments>);
If no target is specified, the result is printed.

In some cases (e.g. `execute, export, keepring, kill, setring, type`) the brackets are optional. In the syntax of the commands `help`, `break`, `quit`, `exit` and `LIB` no brackets are allowed.

5.1.1 attrib

Syntax: `attrib ( name )`

Type: none

Purpose: displays the attribute list of the object called *name*.

Example:

```
ring r=0,(x,y,z),dp;
ideal I=std(maxideal(2));
attrib(I);
↦ attr:isSB, type int
```

Syntax: `attrib ( name , string_expression )`

Type: any

Purpose: returns the value of the attribute *string_expression* of the *name*. If the attribute is not defined for this variable, attrib returns the empty string.

Example:

```
ring r=0,(x,y,z),dp;
ideal I=std(maxideal(2));
attrib(I,"isSB");
↦ 1
// although maxideal(2) is a standard basis,
// SINGULAR does not know it:
attrib(maxideal(2), "isSB");
↦ 0
```

Syntax: `attrib ( name, string_expression, expression )`

Type: none

Purpose: sets the attribute *string_expression* of the *name* to the value *expression*.

Example:

```
ring r=0,(x,y,z),dp;
ideal I=maxideal(2); // the attribute "isSB" is not set
vdim(I);
↦ // ** I is no standardbasis
```

```
↦ 4
attrib(I,"isSB",1);  // the standard basis attribute is set here
vdim(I);
↦ 4
```

Remark: An attribute may be described by any string_expression, but some are reserved. Only the reserved attributes are used by the kernel of SINGULAR. Non-reserved attributes may be used, however, in procedures and can considerably speed up computations.

Reserved attributes:
(not all are used at the moment)

`isSB`	standard basis - set by all commands compution a standard basis like `groebner`, `std`, `stdhilb` etc.; used by `lift`, `dim`, `degree`, `mult`, `hilb`, `vdim`, `kbase`
`isHomog`	the weight vector for homogeneous or quasihomogeneous ideals/modules
`isCI`	complete intersection property
`isCM`	Cohen-Macaulay property
`rank`	set the rank of a module (see Section 5.1.75 [nrows], page 149)
`withSB`	value of type ideal resp. module is std
`withHilb`	value of type intvec is hilb(_,1) (see Section 5.1.39 [hilb], page 125)
`withRes`	value of type list is a free resolution
`withDim`	value of type int is the dimension (see Section 5.1.19 [dim], page 114)
`withMult`	value of type int is the multiplicity (see Section 5.1.69 [mult], page 145)

5.1.2 bareiss

Syntax: `bareiss (` matrix_expression `)`

Type: list of matrix and intvec

Purpose: applies Gauss-Bareiss algorithm (see Section C.5 [References], page 248) to a matrix with an 'optimal' pivotstrategy. Result is a list: the first entry is a lower triangular matrix, the second entry an intvec with the permutations of the columns w.r.t. the original matrix.

Example:

```
ring r2=0,(x(1..9)),ds;
matrix m[4][5]=maxideal(1),maxideal(1);
print(m);
↦ x(1),x(2),x(3),x(4),x(5),
↦ x(6),x(7),x(8),x(9),x(1),
↦ x(2),x(3),x(4),x(5),x(6),
↦ x(7),x(8),x(9),0,   0
list lb=bareiss(m);
print(lb[1]);
↦ _[1,1],_[1,2],0,        0,        0,
↦ _[2,1],_[2,2],_[2,3],   0,        0,
↦ _[3,1],_[3,2],x(6)*x(9),x(5)*x(9),0,
```

```
↦ x(7),  x(8),  0,        0,        x(9)
lb[2];
↦ 1,2,5,4,3
lb[1][3,2];
↦ -x(4)*x(8)+x(3)*x(9)
```

See Section 5.1.17 [det], page 113; Section 4.9 [matrix], page 80.

5.1.3 betti

Syntax: `betti ( list_expression )`
`betti ( resolution )`

Type: intmat

Purpose: computes the graded Betti numbers of R^n/M, if R denotes the basering and M a homogeneous submodule of R^n and the argument represents a resolution of R^n/M: The entry d of the intmat at place (i,j) is the minimal number of generators in degree i+j of the j-th syzygy module (= module of relations) of R^n/M (the 0-th (resp.1-st) syzygy module of R^n/M is R^n (resp. M)). The argument is considered to be the result of a res/sres/mres/lres command. (Only the initial monomials are considered for the computation of the graded Betti numbers.)

Example:

```
ring r=32003,(a,b,c,d),dp;
ideal j=bc-ad,b3-a2c,c3-bd2,ac2-b2d;
list T=mres(j,0); // 0 forces a full resolution
// a minimal set of generators for j:
print(T[1]);
↦ bc-ad,
↦ c3-bd2,
↦ ac2-b2d,
↦ b3-a2c
// second syzygy module of r/j which is the first
// syzygy module of j (minimal generating set):
print(T[2]);
↦ bd,c2,ac,b2,
↦ -a,-b,0, 0,
↦ c, d, -b,-a,
↦ 0, 0, -d,-c
// the second syzygy module (minimal generating set):
print(T[3]);
↦ -b,
↦ a,
↦ -c,
↦ d
print(T[4]);
↦ 0
betti(T);
↦ 1,0,0,0,
↦ 0,1,0,0,
↦ 0,3,4,1
```

```
// most useful for reading off the graded Betti numbers:
print(betti(T),"betti");
↦            0     1     2     3
↦ ------------------------------
↦     0:     1     0     0     0
↦     1:     0     1     0     0
↦     2:     0     3     4     1
↦ ------------------------------
↦ total:     1     4     4     1
```

Hence

- the 0th syzygy module of r/j (which is r) has 1 generator in degree 0 (which is 1)
- the 1st syzygy module `T[1]` (which is j) has 4 generators (one in degree 2 and three in degree 3)
- the 2nd syzygy module `T[2]` has 4 generators (all in degree 4)
- the 3rd syzygy module `T[3]` has 1 generator in degree 5

where the generators are the columns of the shown matrix and degrees are assigned such that the corresponding maps have degree 0 as is shown in the resolution below:

$$0 \longleftarrow r/j \longleftarrow r(1) \xleftarrow{T[1]} r(2) \oplus r^3(3) \xleftarrow{T[2]} r^4(4) \xleftarrow{T[3]} r(5) \longleftarrow 0 \quad .$$

See Section 5.1.59 [lres], page 139; Section 5.1.67 [mres], page 144; Section 5.1.87 [print], page 156; Section 5.1.96 [res], page 162; Section 5.1.104 [sres], page 167; Section C.3 [Syzygies and resolutions], page 247; Section 4.15 [resolution], page 94.

5.1.4 char

Syntax: `char (` ring_name `)`

Type: int

Purpose: returns the characteristic of the coefficient field of a ring.

Example:

```
ring r= 32003,(x,y),dp;
char(r);
↦ 32003
ring s= 0,(x,y),dp;
char(s);
↦ 0
ring ra=(7,a),(x,y),dp;
minpoly=a^3+a+1;
char(ra);
↦ 7
ring rp=(49,a),(x,y),dp;
char(rp);
↦ 7
```

```
ring rr=real,x,dp;
char(rr);
↦ 0
```

See Section 4.16 [ring], page 95; Section 5.1.6 [charstr], page 106.

5.1.5 char_series

Syntax: `char_series ( ideal_expression )`

Type: matrix

Purpose: the rows of the matrix represent the irreducible characteristic series of the ideal with respect to the current variable ordering.
One applicatation is the decomposition of the zero set.

Example:

```
ring r= 32003,(x,y,z),dp;
print(char_series(ideal(xyz,xz,y)));
↦ y,z,
↦ x,y
```

See Section C.4 [Characteristic sets], page 247.

5.1.6 charstr

Syntax: `charstr ( ring_name )`

Type: string

Purpose: returns the description of the coefficient field of a ring.

Example:

```
ring r= 32003,(x,y),dp;
charstr(r);
↦ 32003
ring s= 0,(x,y),dp;
charstr(s);
↦ 0
ring ra=(7,a),(x,y),dp;
minpoly=a^3+a+1;
charstr(ra);
↦ 7,a
ring rp=(49,a),(x,y),dp;
charstr(rp);
↦ 49,a
ring rr=real,x,dp;
charstr(rr);
↦ real
```

See Section 4.16 [ring], page 95; Section 5.1.4 [char], page 105; Section 5.1.80 [ordstr], page 154; Section 5.1.117 [varstr], page 176.

5.1.7 cleardenom

Syntax: `cleardenom ( poly_expression )`
`cleardenom ( vector_expression )`

Type: same as the input type

Purpose: multiplies a polynomial resp. vector by a suitable constant to cancel all denominators from its coefficients and then divide it by its content.

Note:

Example:

```
ring r=0,(x,y,z),dp;
poly f=(3x+6y)^5;
f;
↦ 243x5+2430x4y+9720x3y2+19440x2y3+19440xy4+7776y5
cleardenom(f);
↦ x5+10x4y+40x3y2+80x2y3+80xy4+32y5
```

5.1.8 close

Syntax: `close ( link_expression );`

Type: none

Purpose: closes a link.

Example:

```
link l="MPtcp:launch";
open(l); // start SINGULAR "server" on localhost in batchmode
close(l); // shut down SINGULAR server
```

See Section 4.6 [link], page 66; Section 5.1.77 [open], page 150.

5.1.9 coef

Syntax: `coef ( poly_expression, product_of_ringvars )`

Type: matrix

Syntax: `coef ( vector_expression, product_of_ringvars, matrix_name, matrix_name )`

Type: none

Purpose: determines the monomials in f divisible by one of the ring variables of m (where f is the first argument and m the second argument) and the coefficients of these monomails as polynomials in the remaining variables.

First case: returns a 2 x n matrix M, n being the number of the determined monomials. The first row consists of these monomials, the second row of the corresponding coefficients of the monomials in f. Thus f = M[1,1]*M[2,1]+..+M[1,n]*M[2,n].

Second case: the first matrix (i.e. the 3rd argument) contains the monomials, the second matrix (i.e. the 4th argument) the corresponding coefficients of the monomials in the vector.

Note: coef considers only monomials which really occur in f (i.e. are not 0), while coeffs returns the coefficient 0 at the appropriate place if a monomial is not present (c.f. Section 5.1.10 [coeffs], page 108).

Example:

```
ring r=32003,(x,y,z),dp;
poly f=x5+5x4y+10x2y3+y5;
matrix m=coef(f,y);
print(m);
↦ y5,y3,  y,  1,
↦ 1, 10x2,5x4,x5
f=x^20+xyz+xy+x2y+z3;
print(coef(f,xy));
↦ x20,x2y,xy, 1,
↦ 1,  1,  z+1,z3
vector v=[f,zy+77+xy];
print(v);
↦ [x20+x2y+xyz+z3+xy,xy+yz+77]
matrix mc; matrix mm;
coef(v,y,mc,mm);
print(mc);
↦ x2+xz+x,x20+z3,
↦ x+z,    77
print(mm);
↦ y,1,
↦ y,1
```

See Section 5.1.10 [coeffs], page 108.

5.1.10 coeffs

Syntax:
`coeffs` (poly_expression , ring_variable)
`coeffs` (ideal_expression, ring_variable)
`coeffs` (vector_expression, ring_variable)
`coeffs` (module_expression, ring_variable)
`coeffs` (poly_expression, ring_variable, matrix_name)
`coeffs` (ideal_expression, ring_variable, matrix_name)
`coeffs` (vector_expression, ring_variable, matrix_name)
`coeffs` (module_expression, ring_variable, matrix_name)

Type: matrix

Purpose: developes each polynomial from the first argument, say `J`, (poly_expression/ .. /module_expression) as a univariate polynomial in the given ring_variable, say `z`, and returns the coefficients as a `k` x `d` matrix `M`, where:
`d-1` = maximum `z`-degree of all occuring polynomials
`k` = 1 in case J is a polynomial resp. number of generators in case J is an ideal; in case of a vector or a module this procedure is repeated for each component and the resulting matrices are appended.
If a third argument is present, say `T`, it contains the coefficients such that
`matrix(J) = T*M`.
Thus if $M = (m_{ij})$ the j-th generator of an ideal `J` is equal to

$J_j = z^0 \cdot m_{1j} + z^1 \cdot m_{2j} + ... + z^{d-1} \cdot m_{dj}$,

while for a module J the i-th component of the j-th generator is equal to the entry [i,j] of matrix(J) and we get
$[i,j] = z^0 \cdot m_{(i-1)d+1,j} + z^1 \cdot m_{(i-1)d+2,j} + ... + z^{d-1} \cdot m_{id,j}$

Note: coeffs returns the coefficient 0 at the appropriate place if a monomial is not present, while coef considers only monomials which really occur in the given expression.

Example:

```
ring r;
poly f=(x+y)^5;
matrix M=coeffs(f,y);
print(M);
↦ x5,
↦ 5x4,
↦ 10x3,
↦ 10x2,
↦ 5x,
↦ 1
ideal i=f,xyz+z^10*y^7;
print(coeffs(i,y));
↦ x5,  0,
↦ 5x4, xz,
↦ 10x3,0,
↦ 10x2,0,
↦ 5x,  0,
↦ 1,   0,
↦ 0,   0,
↦ 0,   z10
```

Syntax: `coeffs` (ideal_expression, ideal_expression, product_of_ringvars)
`coeffs` (module_expression, module_expression, product_of_ringvars)

Type: matrix

Purpose: let the first argument be M, the second argument be K (a set of monomials in P), the third argument be the product P of variables to consider. M is supposed to be consisting of elements of (resp. have entries in) a finitely generated module over a ring in the variables not appearing in P. K should contain the generators of the module over this smaller ring. Then `coeffs(M,K,P)` returns a matrix A of coefficients with K*A=M such that the entries of A do not contain any variable from P.
If K does not contain all generators that are necessary to express M, then K*A=M' where M' is the part of M that can be expressed.

Example:

```
ring r=32003,(x,y,z),dp;
ideal M=x2z+y3,xy;
print(coeffs(M,ideal(x2,xy,y2),xy));
↦ z,0,
↦ 0,1,
↦ 0,0
```

See Section 5.1.9 [coef], page 107; Section 5.1.48 [kbase], page 132.

5.1.11 contract

Syntax: `contract ( ideal_expression , ideal_expression )`

Type: matrix

Purpose: contracts each of the n elements of the second ideal J by each of the m elements of the first ideal I, producing a m x n matrix.
Contraction is defined on monomials by:

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

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

Example:

```
ring r=0,(a,b,c,d),dp;
ideal I=a2,a2+bc,abc;
ideal J=a2-bc,abcd;
print(contract(I,J));
↦ 1,0,
↦ 0,ad,
↦ 0,d
```

See Section 5.1.18 [diff], page 113.

5.1.12 dbprint

Syntax: `dbprint ( int_expression , expression_list )`

Type: none

Purpose: applies the print command to each expression in the expression_list, if int_expression is positive. `dbprint` may also be used in procedures in order to print results subject to certain conditions.

Syntax: `dbprint ( expression )`

Type: none

Purpose: The print command is applied to the expression, if `printlevel=>voice`.

Example:

```
int debug=0;
intvec i=1,2,3;
dbprint(debug,i);
debug=1;
dbprint(debug,i);
↦ 1,2,3
voice;
↦ 1
printlevel;
↦ 0
dbprint(i);
```

See Section 5.1.87 [print], page 156; Section 5.3.11 [voice], page 189; Section 5.3.6 [printlevel], page 186; Section 3.8 [Debugging tools], page 46.

5.1.13 defined

Syntax: `defined ( name )`

Type: int

Purpose: returns a value !=0 (TRUE) if there is a userdefined object with this name, and 0 (FALSE) otherwise.
A nonzero return value is the level where the object is defined (level 1 denotes the top level, level 2 the level of a first procedure, level 3 the level of a procedure called by a first procedure, etc.). A local object `m` may be identified by `if (defined(m)==voice)`.

Example:

```
ring r= (0,t),(x,y),dp;
matrix m[5][6]= x,y,1,2,0,x+y;
defined(mm);
↦ 0
defined(r) and defined(m);
↦ 1
defined(m)==voice;              // m is defined in the current level
↦ 1
defined(x);
↦ -1
defined(z);
↦ 0
defined(t);
↦ -1
defined(42);
↦ -1
```

See Section 5.1.99 [rvar], page 164; Section 5.3.11 [voice], page 189.

5.1.14 deg

Syntax: `deg ( poly_expression )`
`deg ( vector_expression )`

Type: int

Purpose: returns the maximal (weighted) degree of the terms of a polynomial or a vector; deg(0) is -1.

Example:

```
ring r=0,(x,y,z),lp;
deg(0);
↦ -1
deg(x3+y4+xyz3);
↦ 5
ring rr=7,(x,y),wp(2,3);
```

```
poly f=x2+y3;
deg(f);
↦ 9
ring R=7,(x,y),ws(2,3);
poly f=x2+y3;
deg(f);
↦ 9
vector v=[x2,y];
deg(v);
↦ 4
```

See Section 5.1.47 [jet], page 131; Section 5.1.79 [ord], page 153; Section 4.12 [poly], page 89; Section 4.18 [vector], page 99.

5.1.15 degree

Syntax: `degree` (ideal_expression)
`degree` (module_expression)

Type: none

Purpose: computes the (Krull) dimension, codimension and the multiplicity of the ideal resp. module generated by the leading monomials of the input and prints it. This is equal to the dimension, codimension and multiplicity of the ideal resp. module if the input is a standard basis with respect to a degree ordering.

Example:

```
ring r3=32003,(x,y,z),ds;
int a,b,c,t =11,10,3,1;
poly f =x^a+y^b+z^(3*c)+x^(c+2)*y^(c-1)+x^(c-1)*y^(c-1)*z3
  +x^(c-2)*y^c*(y2+t*x)^2;
ideal i= jacob(f);
ideal i0=std(i);
degree(i0);
↦ //codimension 3
↦ //dimension 0
↦ //multiplicity 314
```

See Section 4.2 [ideal], page 50; Section 5.1.106 [std], page 169; Section 5.1.19 [dim], page 114; Section 5.1.118 [vdim], page 176; Section 5.1.69 [mult], page 145.

5.1.16 delete

Syntax: `delete` (list_expression, int_expression)

Type: list

Purpose: deletes the element with the given index from a list (the input is not changed).

Example:

```
list l="a","b","c";
list l1=delete(l,2);l1;
↦ [1]:
↦    a
↦ [2]:
↦    c
l;
↦ [1]:
↦    a
↦ [2]:
↦    b
↦ [3]:
↦    c
```

See Section 4.7 [list], page 74; Section 5.1.43 [insert], page 128.

5.1.17 det

Syntax: `det ( intmat_expression )`
`det ( matrix_expression )`

Type: int resp. poly

Purpose: returns the determinant of a square matrix.

Example:

```
ring r=7,(x,y),wp(2,3);
matrix m[3][3]=1,2,3,4,5,6,7,8,x;
det(m);
↦ -3x-1
```

See Section 4.4 [intmat], page 60; Section 4.9 [matrix], page 80; Section 5.1.63 [minor], page 141.

5.1.18 diff

Syntax: `diff ( poly_expression, ring_variable )`
`diff ( vector_expression, ring_variable )`
`diff ( ideal_expression, ring_variable )`
`diff ( module_expression, ring_variable )`
`diff ( matrix_expression, ring_variable )`

Type: the same as the type of the first argument

Syntax: `diff ( ideal_expression, ideal_expression )`

Type: matrix

Purpose: computes the partial derivative of a polynomial object by a ring variable (first forms) respectively differentiates each polynomial (1..n) of the second ideal by the differential operator corresponding to each polynomial (1..m) in the first ideal, producing a m x n matrix.

Example:

```
ring r= 0,(x,y,z),dp;
poly f=2*x^3*y+3*z^5;
diff(f,x);
↦ 6x2y
diff(f,z);
↦ 15z4
vector v=[f,y2+z];
diff(v,z);
↦ 15z4*gen(1)+gen(2)
// corresponds to differential operators
// d2/dx2, d2/dx2+d2/dxdy, d3/dx/dy/dz:
ideal i=x2,x2+yz,xyz;
ideal j=x2-yz,xyz;
print(diff(i,j));
↦ 2,0,
↦ 1,x,
↦ 0,1
```

See Section 4.12 [poly], page 89; Section 4.18 [vector], page 99; Section 4.2 [ideal], page 50; Section 4.10 [module], page 83; Section 4.9 [matrix], page 80; Section 5.1.11 [contract], page 110; Section 5.1.46 [jacob], page 130; Section 5.1.116 [var], page 176;

5.1.19 dim

Syntax: `dim ( ideal_expression )`
`dim ( modul_expression )`

Type: int

Purpose: computes the dimension of the ideal resp. module generated by the leading monomials of the generators representing the given ideal resp. module. This is also the dimension of the ideal if it is represented by a standard basis. Note that the dimension of an ideal I means the Krull dimension of the basering modulo I.
The dimension of a module is the dimension of its annihilator ideal.

Example:

```
ring r=32003,(x,y,z),dp;
ideal I=std(ideal(x2,xy,y5));
dim(I);
↦ 1
```

See Section 4.2 [ideal], page 50; Section 5.1.106 [std], page 169; Section 5.1.15 [degree], page 112; Section 5.1.118 [vdim], page 176; Section 5.1.69 [mult], page 145.

5.1.20 dump

Syntax: `dump ( link_expression );`

Type: none

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

Example:

```
ring r;
// write the whole session to the file dump.ascii
// in ASCII format
dump(":w dump.ascii");
kill r;                  // kill the basering
// reread the session from the file
// redefining everything which was not explicitly killed before
getdump("dump.ascii");
↦ // ** redefining stdfglm **
↦ // ** redefining stdhilb **
↦ // ** redefining groebner **
↦ // ** redefining res **
↦ // ** redefining quot **
↦ // ** redefining quot1 **
↦ // ** redefining quotient0 **
↦ // ** redefining quotient1 **
↦ // ** redefining quotient2 **
↦ // ** redefining quotient3 **
↦ // ** redefining quotient5 **
↦ // ** redefining quotient4 **
↦ // ** redefining intersect1 **
r;
↦ //   characteristic : 32003
↦ //   number of vars : 3
↦ //        block   1 : ordering dp
↦ //                  : names    x y z
↦ //        block   2 : ordering C
```

Restrictions:

For ASCII links, integer matrices contained in lists are dumped as integer list elements (and not as integer matrices), and lists of list are dumped as one flatted list. Furthermore, links themselves are not dumped.

See Section 5.1.36 [getdump], page 124; Section 4.6 [link], page 66; Section 5.1.121 [write], page 178.

5.1.21 eliminate

Syntax: `eliminate` (ideal_expression, product_of_ring_variables)
`eliminate` (module_expression, product_of_ring_variables)
`eliminate` (ideal_expression, product_of_ring_variables, intvec_hilb)
`eliminate` (module_expression, product_of_ring_variables, intvec_hilb)

Type: the same as the type of the first argument

Purpose: eliminates variables occurring as factors of the second argument from an ideal resp. module by intersection with the subring not containing these variables.
`eliminate` does not need a special ordering nor a standard basis as input.

Since elimination is expensive, it might be useful if the input is homogeneous, first to compute the Hilbert function of the ideal (first argument) with a fast ordering (e.g. dp). Then make use of it to speed up the computation: a Hilbert-driven elimination uses the intvec provided as the third argument.

Example:

```
ring r=32003,(x,y,z),dp;
ideal i=x2,xy,y5;
eliminate(i,x);
↦ _[1]=y5
ring R=0,(x,y,t,s,z),dp;
ideal i=x-t,y-t2,z-t3,s-x+y3;
eliminate(i,ts);
↦ _[1]=y2-xz
↦ _[2]=xy-z
↦ _[3]=x2-y
intvec v=hilb(std(i),1);
eliminate(i,ts,v);
↦ _[1]=y2-xz
↦ _[2]=xy-z
↦ _[3]=x2-y
```

See Section 4.2 [ideal], page 50; Section 4.10 [module], page 83; Section 5.1.106 [std], page 169; Section 5.1.39 [hilb], page 125; Section 4.2 [ideal], page 50; Section 4.10 [module], page 83.

5.1.22 eval

Syntax: `eval` (expression)

Type: none

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

Example:

```
link l="MPfile:w example.mp";
ring r=0,(x,y,z),ds;
ideal i=maxideal(3);
ideal j=x7,x2,z;
// compute i+j before writing, but not std
// this writes 'std(ideal(x3,...,z))'
write (l, quote(std(eval(i+j))));
option(prot);
close(l);
// now read it in again and evaluate
// read(l) forces to compute 'std(ideal(x3,...,z))'
read(l);
close(l);
```

See Section 4.6.5.1 [MPfile links], page 69; Section 5.1.90 [quote], page 158; Section 5.1.121 [write], page 178.

5.1.23 execute

Syntax: `execute`(string_expression);

Type: none

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

Note: The string is inserted into the input stream at the next line break, *NOT* after the ; ! Hence, `execute` should be the only command on a line. `execute` should be avoided in procedures whenever possible since it may give rise to name conflicts. Moreover, such procedures cannot be precompiled (something SINGULAR will provide in the future).

Example:

```
ring r=32003,(x,y,z),dp;
ideal i=x+y,z3+22y;
write(">save_i",i);
ring r0=0,(x,y,z),Dp;
string s="ideal k="+read("save_i")+";";
s;
↦ ideal k=x+y,
↦ z3+22y
↦ ;
execute(s); // define the ideal k
k;
↦ k[1]=x+y
↦ k[2]=z3+22y
```

5.1.24 exit

Syntax: `exit;`

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

5.1.25 extgcd

Syntax: `extgcd` (int_expression, int_expression)
`extgcd` (poly_expression, poly_expression)

Type: list of 3 objects of the same type as the type of the arguments

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

Note: polynomials must be univariate to apply `extgcd`

Example:

```
extgcd(24,10);
↦ [1]:
↦    2
↦ [2]:
↦    -2
↦ [3]:
↦    5
ring r=0,(x,y),lp;
extgcd(x4-x6,(x2+x5)*(x2+x3));
↦ [1]:
↦    2x5+2x4
↦ [2]:
↦    x2+x+1
↦ [3]:
↦    1
```

See Section 5.1.34 [gcd], page 123; Section 4.3 [int], page 54; Section 4.12 [poly], page 89

5.1.26 facstd

Syntax: `facstd ( ideal_expression )`
`facstd ( ideal_expression , ideal_expression )`

Type: list of ideals

Purpose: returns a list of Groebner bases by the factorizing Groebner basis algorithm.
The intersection of these ideals has the same zero set as the input, i.e. the radical of the intersection coincides with the radical of the input ideal. In many cases (but not all!) this is already a decomposition of the radical of the ideal. (Note however, that, in general, no inclusion holds.)
The second, optional argument gives a list of polynomials which define non-zero constraints. Hence, the intersection of the output ideals 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.

Note: Not implemented for basering real, galoisfields with gftables and algebraic extensions over the rational numbers.

Example:

```
ring r= 32003,(x,y,z),(c,dp);
ideal I=xyz,x2z;
facstd(I);
↦ [1]:
↦    _[1]=x
↦ [2]:
↦    _[1]=z
facstd(I,x);
↦ [1]:
↦    _[1]=z
```

See Section 5.1.106 [std], page 169; Section 4.16 [ring], page 95; Section 4.2 [ideal], page 50.

5.1.27 factorize

Syntax: `factorize ( poly_expression )`
`factorize ( poly_expression, 0 )`
`factorize ( poly_expression, 2 )`

Type: list of ideal and intvec

Syntax: `factorize ( poly_expression, 1 )`

Type: ideal

Purpose: computes the irreducible factors (as an ideal) of the polynomial together with or without the multiplicities (as an intvec) depending on the second argument:
0: returns factors and multiplicities, first factor is a constant. May also be written with only one argument
1: returns non-constant factors (no multiplicities)
2: returns non-constant factors and multiplicities

Note: Not implemented for the coefficient fields real, finite fields of the type `(p^n, a)`, and algebraic extensions over the rational numbers.

Example:

```
ring r=32003,(x,y,z),dp;
factorize(9*(x-1)^2*(y+z));
↦ [1]:
↦    _[1]=9
↦    _[2]=y+z
↦    _[3]=x-1
↦ [2]:
↦    1,1,2
factorize(9*(x-1)^2*(y+z),1);
↦ _[1]=y+z
↦ _[2]=x-1
factorize(9*(x-1)^2*(y+z),2);
↦ [1]:
↦    _[1]=y+z
↦    _[2]=x-1
↦ [2]:
↦    1,2
```

See Section 4.12 [poly], page 89

5.1.28 fetch

Syntax: `fetch ( ring_name, name )`

Type: number, poly, vector, ideal, module, matrix or list (the same type as the second argument)

Purpose: maps objects between "almost identical" rings. `fetch` is the identity map between rings and qrings with the same coefficient field and the same number of variables (but possibly with different orderings), the i-th variable of the source ring is mapped to the

i-th variable of the basering. This offers a convenient way to change variable names or orderings, or to map objects from a ring to a quotient ring of that ring or vice versa. Compared with `imap`, `fetch` is much more efficient, but less general.

Example:

```
ring r=0,(x,y,z),dp;
ideal i=maxideal(3);
ideal j=std(i);
poly f=x+y2+z3;
vector v=[f,1];
qring q=j;
poly f=fetch(r,f);
f;
↦ z3+y2+x
vector v=fetch(r,v);
v;
↦ z3*gen(1)+y2*gen(1)+x*gen(1)+gen(2)
ideal i=fetch(r,i);
i;
↦ i[1]=z3
↦ i[2]=yz2
↦ i[3]=y2z
↦ i[4]=y3
↦ i[5]=xz2
↦ i[6]=xyz
↦ i[7]=xy2
↦ i[8]=x2z
↦ i[9]=x2y
↦ i[10]=x3
ring rr=0,(a,b,c),lp;
poly f=fetch(q,f);
f;
↦ a+b2+c3
vector v=fetch(r,v);
v;
↦ a*gen(1)+b2*gen(1)+c3*gen(1)+gen(2)
ideal k=fetch(q,i);
k;
↦ k[1]=c3
↦ k[2]=bc2
↦ k[3]=b2c
↦ k[4]=b3
↦ k[5]=ac2
↦ k[6]=abc
↦ k[7]=ab2
↦ k[8]=a2c
↦ k[9]=a2b
↦ k[10]=a3
```

See Section 5.1.41 [imap], page 127; Section 4.8 [map], page 78; Section 4.16 [ring], page 95; Section 4.14 [qring], page 93.

5.1.29 fglm

Syntax: `fglm ( ring_name, ideal_name )`

Type: ideal

Purpose: calculates for the given ideal in the given ring a Groebner basis in the current ring. Implements the so-called FGLM (Faugere, Gianni, Lazard, Mora) algorithm.
The ideal must be zero dimensional and given as a reduced Groebner basis in the given ring. The result is a reduced Groebner basis.
The only permissable difference between the given ring and the current ring may be the monomial ordering and a permutation of the variables resp. parameters.
The main application is to compute a lexicographical Groebner basis from a reduced Groebner basis with respect to a degree ordering. This can be much faster than computing a lexicographical Groebner basis directly.

```
ring r=0,(x,y,z),dp;
ideal i=y3+x2, x2y+x2, x3-x2, z4-x2-y;
option(redSB);   // force the computation of a reduced SB
i=std(i);
vdim(i);
↦ 28
ring s=0,(z,x,y),lp;
ideal j=fglm(r,i);
j;
↦ j[1]=y4+y3
↦ j[2]=xy3-y3
↦ j[3]=x2+y3
↦ j[4]=z4+y3-y
```

See Section 4.16 [ring], page 95; Section 4.14 [qring], page 93; Section 5.1.106 [std], page 169; Section 5.1.78 [option], page 150; Section 5.1.118 [vdim], page 176.

5.1.30 input from files

Syntax: `< "'filename'";`

Type: none

Purpose: input comes from new file '`filename`'. This is shorthand `execute read(filename)`.

Example:

```
< "example"; //read in the file example and execute it
```

See Section 5.1.23 [execute], page 117; Section 5.1.93 [read], page 160.

5.1.31 find

Syntax: `find ( string_expression, substring_expression )`
`find ( string_expression, substring_expression, int_expression )`

Type: int

Purpose: returns the first position of the substring in the string or 0 (if not found), start the search at the position given in the 3rd argument.

Example:

```
find("Aac","a");
↦ 2
find("abab","a"+"b");
↦ 1
find("abab","a"+"b",2);
↦ 3
find("abab","ab",3);
↦ 3
find("0123","abcd");
↦ 0
```

See Section 4.17 [string], page 96.

5.1.32 finduni

Syntax: `finduni ( ideal_expression )`

Type: ideal

Purpose: returns an ideal such that the it-h generator is an univariate polynomial in the it-h ring variable belonging to ideal_expression.
The polynomials have minimal degree w.r.t. this property. The ideal must be zero dimensional and given as a reduced Groebner basis in the current ring.

Example:

```
ring  r= 0,(x,y,z), dp;
ideal i= y3+x2, x2y+x2, z4-x2-y;
option(redSB);  // force computation of reduced basis
i= std(i);
ideal k= finduni(i);
print(k);
↦ x4-x2,
↦ y4+y3,
↦ z12
```

See Section 4.16 [ring], page 95; Section 5.1.106 [std], page 169; Section 5.1.78 [option], page 150; Section 5.1.118 [vdim], page 176.

5.1.33 freemodule

Syntax: `freemodule ( int_expression )`

Type: module

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

Example:

```
ring r= 32003,(x,y),(c,dp);
freemodule(3);
↦ _[1]=[1]
↦ _[2]=[0,1]
↦ _[3]=[0,0,1]
matrix m = freemodule(3); // generates the 3x3 unit matrix
print(m);
↦ 1,0,0,
↦ 0,1,0,
↦ 0,0,1
```

See Section 5.1.35 [gen], page 123; Section 4.10 [module], page 83.

5.1.34 gcd

Syntax: `gcd ( int_expression, int_expression )`
`gcd ( poly_expression, poly_expression )`

Type: the same as the type of the arguments

Purpose: computes the greatest common divisor.

Note: Not implemented for the coefficient fields real, finite fields of the type `(p^n, a)`, and algebraic extensions over the rational numbers.

Example:

```
gcd(2,3);
↦ 1
ring r=0,(x,y,z),lp;
gcd(3x2*(x+y),9x*(y2-x2));
↦ x2+xy
```

See Section 5.1.25 [extgcd], page 117; Section 4.3 [int], page 54; Section 4.12 [poly], page 89

5.1.35 gen

Syntax: `gen ( int_expression )`

Type: vector

Purpose: returns the i-th free generator of a free module.

Example:

```
ring r= 32003,(x,y,z),(c,dp);
gen(3);
↦ [0,0,1]
vector v=gen(5);
poly f= xyz;
v=v+f*gen(4); v;
↦ [0,0,0,xyz,1]
ring rr= 32003,(x,y,z),dp;
fetch(r,v);
↦ xyz*gen(4)+gen(5)
```

See Section 5.1.33 [freemodule], page 122; Section 4.3 [int], page 54; Section 4.18 [vector], page 99.

5.1.36 getdump

Syntax: `getdump ( link_expression ) ;`

Type: none

Purpose: reads the contents of the entire file resp. link and restores all variables from it. For ASCII links, `getdump` is equivalent to a `execute read ( link );` command. For MP links, `getdump` should only be used on data which was previously `dump`'ed.

Example:

```
int i=3;
dump(":w example.txt");
kill i;
option(noredefine);
getdump("example.txt");
i;
↦ 3
```

Restrictions:

`getdump` is not supported for DBM links, or for a link connecting to `stdin` (standard input).

See Section 4.6 [link], page 66; Section 5.1.20 [dump], page 114; Section 5.1.93 [read], page 160.

5.1.37 groebner

Syntax: `groebner ( ideal_expression )`
`groebner ( module_expression )`

Type: ideal or module

Purpose: returns a standard basis of an ideal or module with respect to the monomial ordering of the basering using a heuristically choosen method.

Example:

```
ring r = 0, (a,b,c,d), lp;
ideal i = a+b+c+d, ab+ad+bc+cd, abc+abd+acd+bcd, abcd-1;
groebner(i);
↦ _[1]=c2d6-c2d2-d4+1
↦ _[2]=c3d2+c2d3-c-d
↦ _[3]=bd4-b+d5-d
↦ _[4]=bc-bd5+c2d4+cd-d6-d2
↦ _[5]=b2+2bd+d2
↦ _[6]=a+b+c+d
```

See Section 5.1.106 [std], page 169; Section 5.1.107 [stdfglm], page 170; Section 5.1.108 [stdhilb], page 171.

5.1.38 help

Syntax: `help;`
`help` command_name ;
`help` section_title ;
`help` lib_name ;
`help` procedure_name ;

Type: none

Purpose: provides help information.
The online version of the manual is used for the first three forms of the `help` command. `help` lib_name ; and `help` procedure_name ; directly display the help sections from libraries and procedures, resp., if those are present. The latter forms do not enter the online help system.

Note: `?` may be used instead of `help`.

Example:

```
help;
help ring;
help Rings and orderings;
help all.lib;
```

See Section 3.7.1 [Format of a library], page 45; Section 3.6.1 [Procedure definition], page 40; Section 3.1.3 [The online help system], page 16.

5.1.39 hilb

Syntax: `hilb (` ideal_expression `)`
`hilb (` module_expression `)`
`hilb (` ideal_expression`,` int_expression `)`
`hilb (` module_expression`,` int_expression `)`

Type: none (if called with one argument)
intvec (if called with two arguments)

Purpose: computes the Hilbert series of the ideal resp. module defined by the leading terms of the generators of the given ideal resp. module. If the input is homogeneous (all variable weights 1) and a standard basis, this is the Hilbert series of the original ideal resp. module.

Prints first and second Hilbert series with one argument, returns the n-th (n=1, 2) Hilbert series as intvec with two arguments.

Example:

```
ring R=32003,(x,y,z),dp;
ideal i=x2,y2,z2;
ideal s=std(i);
hilb(s);
↦ //         1 t^0
↦ //        -3 t^2
↦ //         3 t^4
↦ //        -1 t^6
```

```
↦
↦ //         1 t^0
↦ //         3 t^1
↦ //         3 t^2
↦ //         1 t^3
↦ // codimension = 3
↦ // dimension   = 0
↦ // degree      = 8
hilb(s,1);
↦ 1,0,-3,0,3,0,-1,0
hilb(s,2);
↦ 1,3,3,1,0
```

See Section C.2 [Hilbert function], page 246; Section 4.2 [ideal], page 50; Section 4.5 [intvec], page 63; Section 4.10 [module], page 83; Section 5.1.108 [stdhilb], page 171; Section 5.1.106 [std], page 169.

5.1.40 homog

Syntax: `homog ( ideal_expression )`
`homog ( module_expression )`

Type: int

Purpose: tests for homogeneity: return 1 for homogeneous input, 0 otherwise.

Syntax:

`homog ( polynomial_expression , ring_variable )`
`homog ( vector_expression , ring_variable )`
`homog ( ideal_expression , ring_variable )`
`homog ( module_expression , ring_variable )`

Type: same as first argument

Purpose: homogenizes polynomials, vectors, ideals or modules by multiplying each monomial with a suitable power of the given ring variable (which must have weight 1).

Example:

```
ring r=32003,(x,y,z),ds;
poly s1=x3y2+x5y+3y9;
poly s2=x2y2z2+3z8;
poly s3=5x4y2+4xy5+2x2y2z3+y7+11x10;
ideal i=s1,s2,s3;
homog(s2,z);
↦ x2y2z4+3z8
homog(i,z);
↦ _[1]=3y9+x5yz3+x3y2z4
↦ _[2]=x2y2z4+3z8
↦ _[3]=11x10+y7z3+5x4y2z4+4xy5z4+2x2y2z6
homog(i);
↦ 0
homog(homog(i,z));
↦ 1
```

See Section 4.12 [poly], page 89; Section 4.18 [vector], page 99; Section 4.2 [ideal], page 50; Section 4.10 [module], page 83.

5.1.41 imap

Syntax: `imap ( ring_name, name )`

Type: number, poly, vector, ideal, module, matrix or list (the same type as the second argument)

Purpose: identity map on common subrings. `imap` is the map between rings, and qrings with the compatible ground coefficient field which is the identity on variables and parameters of the same name and 0 otherwise. (See Section 4.8 [map], page 78 for a description of possible mapping between different ground fields). Useful for mapping from a homogenizing ring to the original ring or for mappings from/to rings with/without parameters. Compared with `fetch`, `imap` is more general, but less efficient.

Example:

```
ring r = 0,(x,y,z,a,b,c),dp;
ideal i = xy2z3a4b5 + 1, homog(xy2z3a4b5 + 1, c); i;
↦ i[1]=xy2z3a4b5+1
↦ i[2]=xy2z3a4b5+c15
ring r1 = 0,(a,b,x,y,z),lp;
ideal j = imap(r, i); j;
↦ j[1]=a4b5xy2z3+1
↦ j[2]=a4b5xy2z3
ring r2 = (0, a, b), (x,y,z), ls;
ideal j = imap(r, i); j;
↦ j[1]=(1)+(a4b5)*xy2z3
↦ j[2]=(a4b5)*xy2z3
```

See Section 5.1.28 [fetch], page 119; Section 5.1.40 [homog], page 126; Section 4.8 [map], page 78; Section 4.14 [qring], page 93; Section 4.16 [ring], page 95.

5.1.42 indepSet

Syntax: `indepSet ( ideal_expression )`

Type: intvec

Purpose: computes a maximal set U of independent variables of the ideal given by a standard basis. If v is the result then v[i] is 1 if and only if the i-th variable of the ring is an independent variable. Hence the set U consisting of the variables x(i) s.t. v[i]=1 is a maximal independent set.

U is a set of independent variables if and only if $I \cap K[U] = (0)$ is, i.e., eliminating the remaining variables gives (0).

Syntax: `indepSet ( ideal_expression, int_expression )`

Type: list

Purpose: computes a list of all maximal sets (if the flag is 0), or of all sets of independent variables of the ideal given by the first argument

Example:

```
ring r=32003,(x,y,z,u,v,w),dp;
ideal I=xyzw,yzvw,uzw,xyv;
attrib(I,"isSB",1);
indepSet(I);
↦ 1,1,1,1,0,0
eliminate(I,vw);
↦ _[1]=0
indepSet(I,0);
↦ [1]:
↦    1,1,1,1,0,0
↦ [2]:
↦    1,0,1,1,1,0
↦ [3]:
↦    0,1,1,1,1,0
↦ [4]:
↦    1,1,0,1,0,1
↦ [5]:
↦    1,0,1,0,1,1
↦ [6]:
↦    1,0,0,1,1,1
↦ [7]:
↦    0,1,0,1,1,1
indepSet(I,1);
↦ [1]:
↦    1,1,1,1,0,0
↦ [2]:
↦    1,0,1,1,1,0
↦ [3]:
↦    0,1,1,1,1,0
↦ [4]:
↦    1,1,0,1,0,1
↦ [5]:
↦    1,0,1,0,1,1
↦ [6]:
↦    1,0,0,1,1,1
↦ [7]:
↦    0,1,0,1,1,1
↦ [8]:
↦    0,1,1,0,0,1
eliminate(I,xuv);
↦ _[1]=0
```

See Section 5.1.106 [std], page 169; Section 4.2 [ideal], page 50.

5.1.43 insert

Syntax: `insert` (list_expression, expression)
`insert` (list_expression, expression, int_expression)

Type: list

Purpose: inserts a new element into a list at the beginning or (if called with 3 arguments) after the given position (the input is not changed).

Example:

```
list L=1,2;
insert(L,4,2);
↦ [1]:
↦    1
↦ [2]:
↦    2
↦ [3]:
↦    4
insert(L,4);
↦ [1]:
↦    4
↦ [2]:
↦    1
↦ [3]:
↦    2
```

See Section 4.7 [list], page 74; Section 5.1.16 [delete], page 112.

5.1.44 interred

Syntax: `interred ( ideal_expression )`
`interred ( module_expression )`

Type: the same as the input type

Purpose: interreduces a set of polynomials/vectors.
input: $f_1, ..., f_n$
output: $g_1, ..., g_s$ with $s <= n$ and the properties

- $(f_1, ..., f_n) = (g_1, ..., g_s)$
- $\mathrm{L}(g_i) \neq \mathrm{L}(g_j)$ for all $i \neq j$
- in the case of a global ordering (polynomial ring):
  $L(g_i)$ does not divide m for all monomials m of $\{g_1, ..., g_{i-1}, g_{i+1}, ..., g_s\}$
- in the case of a local or mixed ordering (localization of polynomial ring):
  if, for any i<>j, $L(g_i)|L(g_j)$ then $ecart(g_i) > ecart(g_j)$

Remark: $L(g)$ denotes the leading term of g and $ecart(g) = deg(g) - deg(L(g))$.

Example:

```
ring r=0,(x,y,z),dp;
ideal i=x2+z,z,2z;
interred(i);
↦ _[1]=z
↦ _[2]=x2
ring R=0,(x,y,z),ds;
ideal i=zx+y3,z+y3,z+xy;
interred(i);
```

```
↦ _[1]=z+y3
↦ _[2]=z+xy
↦ _[3]=x2y-y3
```

See Section 4.2 [ideal], page 50; Section 4.10 [module], page 83; Section 5.1.106 [std], page 169.

5.1.45 intersect

Syntax: `intersect` (expression_list of ideal_expression)
`intersect` (expresion_list of module_expression)

Type: ideal resp. module

Purpose: ideal resp. module intersection.

Note: If the option `computeSB` is enabled then the result is a standard basis.

Example:

```
ring R=0,(x,y),dp;
ideal i=x;
ideal j=y;
intersect(i,j);
↦ _[1]=xy
ring r=181,(x,y,z),(c,ls);
ideal id1=maxideal(3);
ideal id2=x2+xyz,y2-z3y,z3+y5xz;
ideal id3=intersect(id1,id2,ideal(x,y));
id3;
↦ id3[1]=yz3+xy6z
↦ id3[2]=yz4-y2z
↦ id3[3]=y2z3-y3
↦ id3[4]=xz3+x2y5z
↦ id3[5]=xyz2+x2z
↦ id3[6]=xyz3-xy2
↦ id3[7]=xy2z+x2y
↦ id3[8]=x2yz+x3
```

See Section 4.2 [ideal], page 50; Section 4.10 [module], page 83; Section 5.1.78 [option], page 150.

5.1.46 jacob

Syntax: `jacob` (poly_expression)
`jacob` (ideal_expression)

Type: ideal, if the input is a polynomial
matrix, if the input is an ideal

Purpose: computes the Jacobi ideal resp. Jacobi matrix generated by all partial derivatives of the input.

Example:

```
ring R;
poly f=x^2+y^3+z^5;
jacob(f);
↦ _[1]=2x
↦ _[2]=3y2
↦ _[3]=5z4
ideal i=jacob(f);
print(jacob(i));
↦ 2,0, 0,
↦ 0,6y,0,
↦ 0,0, 20z3
```

See Section 4.2 [ideal], page 50; Section 4.10 [module], page 83; Section 5.1.18 [diff], page 113; Section 5.1.76 [nvars], page 150.

5.1.47 jet

Syntax:
```
jet ( poly_expression, int_expression )
jet ( vector_expression, int_expression )
jet ( ideal_expression, int_expression )
jet ( module_expression, int_expression )
jet ( poly_expression, int_expression, intvec_expression )
jet ( vector_expression, int_expression, intvec_expression )
jet ( ideal_expression, int_expression, intvec_expression )
jet ( module_expression, int_expression, intvec_expression )
```

Type: the same as the type of the first argument

Purpose: deletes from the first argument all terms of degree (resp. weighted degree where the weights are given by the third argument) bigger than the second argument.

Example:

```
ring r=32003,(x,y,z),(c,dp);
jet(1+x+x2+x3+x4,3);
↦ x3+x2+x+1
poly f=1+x+y+z+x2+xy+xz+y2+x3+y3+x2y2+z4;
jet(f,3);
↦ x3+y3+x2+xy+y2+xz+x+y+z+1
intvec iv=2,1,1;
jet(f,3,iv);
↦ y3+xy+y2+xz+x+y+z+1
// the part of f with (total) degree >3:
f-jet(f,3);
↦ x2y2+z4
// the homogeneous part of f of degree 2:
jet(f,2)-jet(f,1);
↦ x2+xy+y2+xz
// the part of maximal degree:
jet(f,deg(f))-jet(f,deg(f)-1);
↦ x2y2+z4
// the absolute term of f:
jet(f,0);
```

```
↦ 1
// now for other types:
ideal i=f,x,f*f;
jet(i,2);
↦ _[1]=x2+xy+y2+xz+x+y+z+1
↦ _[2]=x
↦ _[3]=3x2+4xy+3y2+4xz+2yz+z2+2x+2y+2z+1
vector v=[f,1,x];
jet(v,1);
↦ [x+y+z+1,1,x]
jet(v,0);
↦ [1,1]
v=[f,1,0];
module m=v,v,[1,x2,z3,0,1];
jet(m,2);
↦ _[1]=[x2+xy+y2+xz+x+y+z+1,1]
↦ _[2]=[x2+xy+y2+xz+x+y+z+1,1]
↦ _[3]=[1,x2,0,0,1]
```

See Section 5.1.14 [deg], page 111; Section 4.3 [int], page 54; Section 4.5 [intvec], page 63; Section 4.2 [ideal], page 50; Section 4.10 [module], page 83; Section 4.12 [poly], page 89; refvector.

5.1.48 kbase

Syntax: `kbase` (ideal_expression)
`kbase` (module_expression)
`kbase` (ideal_expression, int_expresion)
`kbase` (module_expression, int_expresion)

Type: the same as the input type of the first argument

Purpose:

with one argument: computes a vector space basis (consisting of monomials) of the quotient ring by the ideal resp. of a free module by the module in case this is finite dimensional and if the input is a standard basis with respect to the ring ordering. If the input is no standard basis, the leading terms of the input are used and the result may have no meaning.
with two arguments: a part of the same result with degree of the monomials equal to the second argument, the quotient need not be finite dimensional.

Example:

```
ring r=32003,(x,y,z),ds;
ideal i=x2,y2,z;
kbase(std(i));
↦ _[1]=xy
↦ _[2]=y
↦ _[3]=x
↦ _[4]=1
i=x2,y3,xyz;
kbase(std(i),2);
↦ _[1]=z2
↦ _[2]=yz
```

```
↦ _[3]=xz
↦ _[4]=y2
↦ _[5]=xy
```

See Section 4.2 [ideal], page 50; Section 4.10 [module], page 83; Section 5.1.118 [vdim], page 176.

5.1.49 kill

Syntax: `kill`(name);
`kill`(list_of_names);

Type: none

Purpose: deletes objects.

Example:

```
int i=3;
ring r= 0,x,dp;
poly p;
listvar();
↦ // r                    [0]  *ring
↦ //      p                   [0]  poly
↦ // i                    [0]  int 3
↦ // LIB                  [0]  string standard.lib
kill(i,r);
// the variable ‘i‘ does not exist any more
i;
↦    ? ‘i‘ is undefined
↦    ? error occurred in Z line 7: ‘i;‘
listvar();
↦ // LIB                  [0]  string standard.lib
```

See Section 5.1.13 [defined], page 111; Section D.3 [general_lib], page 250; Section 5.1.71 [names], page 146

5.1.50 killattrib

Syntax: `killattrib` (name, string_expression) ;

Type: none

Purpose: deletes the attribute given as the second argument.

Example:

```
ring r= 32003,(x,y),lp;
ideal i=maxideal(1);
attrib(i,"isSB",1);
attrib(i);
↦ attr:isSB, type int
killattrib(i,"isSB");
attrib(i);
↦ no attributes
```

See Section 5.1.1 [attrib], page 102; Section 5.1.78 [option], page 150.

5.1.51 koszul

Syntax: `koszul ( int_expression, int_expression )`
`koszul ( int_expression, ideal_expression )`
`koszul ( int_expression, int_expression, ideal_expression )`

Type: matrix

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

`koszul(d,id)` computes a matrix of the Koszul relations of degree d of the generators of the ideal `id`.

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

koszul(1,id),koszul(2,id),... form a complex, that is, the product of `koszul(i,id)` and `koszul(i+1,id)` equals zero.

Example:

```
ring r=32003,(x,y,z),dp;
print(koszul(3,2));
↦ -y,-z,0,
↦ x, 0, -z,
↦ 0, x, y
ideal I=xz2+yz2+z3,xyz+y2z+yz2,xy2+y3+y2z;
print(koszul(1,I));
↦ xz2+yz2+z3,xyz+y2z+yz2,xy2+y3+y2z
print(koszul(2,I));
↦ -xyz-y2z-yz2,-xy2-y3-y2z,0,
↦ xz2+yz2+z3,  0,          -xy2-y3-y2z,
↦ 0,           xz2+yz2+z3, xyz+y2z+yz2
print(koszul(2,I)*koszul(3,I));
↦ 0,
↦ 0,
↦ 0
```

See Section 4.3 [int], page 54; Section 4.9 [matrix], page 80.

5.1.52 lead

Syntax: `lead ( poly_expression )`
`lead ( vector_expression )`
`lead ( ideal_expression )`
`lead ( module_expression )`

Type: the same as the input type

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

Note: `IN` may be used instead of `lead`.

Example:

```
ring r=32003,(x,y,z),(c,ds);
poly f=x2+y+z3;
vector v=[x^10,f];
ideal i=f,z;
module m=v,[0,0,1+x];
lead(f);
↦ y
lead(v);
↦ [x10]
lead(i);
↦ _[1]=y
↦ _[2]=z
lead(m);
↦ _[1]=[x10]
↦ _[2]=[0,0,1]
lead(0);
↦ 0
```

See Section 4.12 [poly], page 89; Section 4.18 [vector], page 99; Section 4.2 [ideal], page 50; Section 4.10 [module], page 83.

5.1.53 leadcoef

Syntax: `leadcoef ( poly_expression )`
`leadcoef ( vector_expression )`

Type: number

Purpose: returns the leading (or initial) coefficient of a polynomial or a vector with respect to the monomial ordering.

Example:

```
ring r=32003,(x,y,z),(c,ds);
poly f=x2+y+z3;
vector v=[2*x^10,f];
leadcoef(f);
↦ 1
leadcoef(v);
↦ 2
leadcoef(0);
↦ 0
```

See Section 4.11 [number], page 86 Section 4.12 [poly], page 89; Section 4.18 [vector], page 99.

5.1.54 leadexp

Syntax: `leadexp ( poly_expression )`
`leadexp ( vector_expression )`

Type: intvec

Purpose: returns the exponent vector of the leading monomial of a polynomial or a vector.

Example:

```
ring r=32003,(x,y,z),(c,ds);
poly f=x2+y+z3;
vector v=[2*x^10,f];
leadexp(f);
↦ 0,1,0
leadexp(v);
↦ 10,0,0
leadexp(0);
↦ 0,0,0
```

See Section 4.5 [intvec], page 63; Section 4.12 [poly], page 89; Section 4.18 [vector], page 99.

5.1.55 LIB

Syntax: `LIB` string_expression;

Type: none

Purpose: reads a library of procedures from a file. If the given filename does not start with `~`, `.` or `/`, the following directories are searched for (in that order): the current directory, the directories given in the environment variable `SINGULARPATH`, some default directories relative to the location of the SINGULAR executable program, and finally some default absolute directories. You can view the search path which SINGULAR uses to locate its libraries, by starting up SINGULAR with the option `-v`, or by issuing the command `system("with");`".

Only the names of the procedures in the library are loaded, the body of the procedures is read during the first call of this procedure. This minimizes memory consumption by unused procedures. When SINGULAR is started with the `-q` or `--quiet` option, no message about the loading of a library is displayed. More exactly, option `-q` (and likewise `--quiet`) unsets option `loadLib` to inhibit monitoring of library loading (see Section 5.1.78 [option], page 150).

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

Syntax: `LIB;`

Type: string

Purpose: shows all loaded libraries.

Example:

```
option(loadLib); // show loading of libraries; standard.lib is loaded
LIB;
↦ standard.lib
                  // the names of the procedures of inout.lib
LIB "inout.lib"; // are now known to Singular
↦ // ** loaded /home/obachman/Singular-1.2/Singular/LIB/inout.lib (1.6,
↦ 1998/05/14)
LIB;
↦ standard.lib,inout.lib
```

See Section 3.1.6 [Command line options], page 18; Section 2.3.3 [Procedures and libraries], page 10; Appendix D [SINGULAR libraries], page 250; Section 4.13 [proc], page 92; Section D.1 [standard_lib], page 250; Section 4.17 [string], page 96; Section 5.1.110 [system], page 172.

5.1.56 lift

Syntax: `lift ( ideal_expression, subideal_expression )`
`lift ( module_expression, submodule_expression )`

Type: matrix

Purpose: computes the transformation matrix which expresses the generators of a submodule in terms of the generators of a module. Uses different algorithms for modules which are resp. are not represented by a standard basis.
Hence, if `sm` is the submodule (or ideal), `m` the module (or ideal) and `T` the transformation matrix returned by lift then `matrix(sm)=matrix(m)*T` and `sm=module(matrix(m)*T)` (or `sm=ideal(matrix(m)*T)`). Gives a warning if `sm` is not a submodule.

Note: For local or mixed orderings this holds only up to units in the associated localized ring.

Example:

```
ring r;
ideal m=3x2+yz,7y6+2x2y+5xz;
poly f=y7+x3+xyz+z2;
ideal i=jacob(f);
matrix T=lift(i,m);
matrix(m)-matrix(i)*T;
↦ _[1,1]=0
↦ _[1,2]=0
```

See Section 4.2 [ideal], page 50; Section 4.10 [module], page 83.

5.1.57 liftstd

Syntax: `liftstd ( ideal_expression, matrix_name )`
`liftstd ( module_expression, matrix_name )`

Type: ideal or module

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

Example:

```
ring R=0,(x,y,z),dp;
poly f=x3+y7+z2+xyz;
ideal i=jacob(f);
matrix T;
ideal sm=liftstd(i,T);
sm;
```

```
↦ sm[1]=xy+2z
↦ sm[2]=3x2+yz
↦ sm[3]=yz2+3048192z3
↦ sm[4]=3024xz2-yz2
↦ sm[5]=y2z-6xz
↦ sm[6]=3097158156288z4+2016z3
↦ sm[7]=7y6+xz
print(T);
↦ 0,1,T[1,3],   T[1,4],y,  T[1,6],0,
↦ 0,0,-3x+3024z,3x,    0,  T[2,6],1,
↦ 1,0,T[3,3],   T[3,4],-3x,T[3,6],0
matrix(sm)-matrix(i)*T;
↦ _[1,1]=0
↦ _[1,2]=0
↦ _[1,3]=0
↦ _[1,4]=0
↦ _[1,5]=0
↦ _[1,6]=0
↦ _[1,7]=0
```

See Section 4.2 [ideal], page 50; Section 4.16 [ring], page 95; Section 5.1.78 [option], page 150; Section 5.1.106 [std], page 169; Section 4.9 [matrix], page 80.

5.1.58 listvar

Syntax:
```
listvar();
listvar( type );
listvar( ring_name );
listvar( name );
listvar( all );
```

Type: none

Purpose: lists all (user-)defined names:
`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 procedures
The current basering is marked with a `*`. The nesting level of variables in procedures is shown in square brackets.

Example:

```
proc t1 { }
proc t2 { }
ring s;
poly ss;
ring r;
poly f=x+y+z;
int i=7;
ideal I=f,x,y;
listvar(all);
```

```
↦ // i                    [0]  int 7
↦ // r                    [0]  *ring
↦ //      I                     [0]  ideal, 3 generator(s)
↦ //      f                     [0]  poly
↦ // s                    [0]  ring
↦ //      ss                    [0]  poly
↦ // LIB                  [0]  string standard.lib
listvar();
↦ // i                    [0]  int 7
↦ // r                    [0]  *ring
↦ //      I                     [0]  ideal, 3 generator(s)
↦ //      f                     [0]  poly
↦ // s                    [0]  ring
↦ // LIB                  [0]  string standard.lib
listvar(r);
↦ // r                    [0]  *ring
↦ // I                    [0]  ideal, 3 generator(s)
↦ // f                    [0]  poly
listvar(t1);
↦ // t1                   [0]  proc
listvar(proc);
↦ // t2                   [0]  proc
↦ // t1                   [0]  proc
↦ // intersect1          [0]  proc from standard.lib (static)
↦ // quotient4           [0]  proc from standard.lib
↦ // quotient5           [0]  proc from standard.lib
↦ // quotient3           [0]  proc from standard.lib
↦ // quotient2           [0]  proc from standard.lib
↦ // quotient1           [0]  proc from standard.lib
↦ // quotient0           [0]  proc from standard.lib (static)
↦ // quot1               [0]  proc from standard.lib (static)
↦ // quot                [0]  proc from standard.lib
↦ // res                 [0]  proc from standard.lib
↦ // groebner            [0]  proc from standard.lib
↦ // stdhilb             [0]  proc from standard.lib
↦ // stdfglm             [0]  proc from standard.lib
```

See Section 3.6.2 [Names in procedures], page 42; Section 3.4.3 [Names], page 32; Section 5.1.13 [defined], page 111; Section 5.1.71 [names], page 146; Section 5.1.114 [type], page 174.

5.1.59 lres

Syntax: `lres` (ideal_expression , int_expression)
`lres` (module_expression , int_expression)

Type: resolution

Purpose: computes a free resolution of an ideal or module with LaScala's method.
The ideal_expression resp. module_expression has to be homogeneous.

More precisely, let M be given by a a generating set and $A_1 = matrix(M)$. Then `lres` computes a minimal free resolution of $M_1 = coker(A_1)$

$$... \longrightarrow F_2 \xrightarrow{A_2} F_1 \xrightarrow{A_1} F_0 \longrightarrow M_1 \longrightarrow 0,$$

The computation stops after k steps, if the int expression k is not zero, and returns a list of modules M_i = `module`(A_i), i=1..k.

If k=0, `lres(M,0)` returns a list of n modules where n is the number of variables of the basering.

Let `list l=lres(M,0);` then L[1] generates M, L[2] generates the first syzygy module of L[1], etc. (L[i] = M_i in the notations from above).

Note: To access the elements of a resolution, it has to be assigned to a list, which also completes computations and may therefore take time.

Example:

```
ring r=0,(x,y,z),dp;
ideal i=xz,yz,x^3-y^3;
def l=lres(i,0);
print(betti(l),"betti"); // input to betti may be of type resolution
↦            0     1     2
↦ ------------------------
↦     0:     1     0     0
↦     1:     0     2     1
↦     2:     0     1     1
↦ ------------------------
↦ total:     1     3     2
list ll=l;
l[2];                    // elements can only be accessed in the list
↦ _[1]=-x*gen(1)+y*gen(2)
↦ _[2]=-x2*gen(2)+y2*gen(1)+z*gen(3)
```

See Section 5.1.3 [betti], page 104; Section 4.2 [ideal], page 50; Section 4.3 [int], page 54; Section 5.1.64 [minres], page 142; Section 4.10 [module], page 83; Section 5.1.67 [mres], page 144; Section 5.1.96 [res], page 162; Section 5.1.104 [sres], page 167.

5.1.60 maxideal

Syntax: `maxideal (` int_expression `)`

Type: ideal

Purpose: returns the power given by int_expression of the maximal ideal generated by all variables (maxideal(i)=(1) for i<=0).

Example:

```
ring r=32003,(x,y,z),dp;
maxideal(2);
↦ _[1]=z2
↦ _[2]=yz
↦ _[3]=y2
↦ _[4]=xz
↦ _[5]=xy
↦ _[6]=x2
```

See Section 4.2 [ideal], page 50; Section 4.16 [ring], page 95.

5.1.61 memory

Syntax: `memory ( int_expression )`

Type: int

Purpose: returns statistics concerning the memory management:
`memory(0)` is the number of active bytes
`memory(1)` is the number of allocated bytes

Example:

```
"Objects of SINGULAR use (at the moment) ",memory(0)," bytes,"
 +newline+
"allocated from system (at the moment):", memory(1), "bytes";
↦ Objects of SINGULAR use (at the moment)  107290  bytes,
↦ allocated from system (at the moment): 110594 bytes
```

See Section 5.1.78 [option], page 150.

5.1.62 minbase

Syntax: `minbase ( ideal_expression )`
`minbase ( module_expression )`

Type: the same as the type of the argument

Purpose: returns a minimal set of generators of an ideal resp. module if the input is either homogeneous or if the ordering is local.

Example:

```
ring r=181,(x,y,z),(c,ls);
ideal id2=x2+xyz,y2-z3y,z3+y5xz;
ideal id4=maxideal(3)+id2;
size(id4);
↦ 13
minbase(id4);
↦ _[1]=x2
↦ _[2]=xyz+x2
↦ _[3]=xz2
↦ _[4]=y2
↦ _[5]=yz2
↦ _[6]=z3
```

See Section 5.1.68 [mstd], page 145.

5.1.63 minor

Syntax: `minor ( matrix_expression , int_expression )`

Type: ideal

Purpose: returns the set of all minors (=subdeterminants) of the given size of a matrix.

Example:

```
ring r = 0, (x(1..5)), ds;
matrix m[2][4]= x(1..4), x(2..5);
print(m);
↦ x(1),x(2),x(3),x(4),
↦ x(2),x(3),x(4),x(5)
ideal j = minor(m, 2);
j;
↦ j[1]=-x(4)^2+x(3)*x(5)
↦ j[2]=-x(3)*x(4)+x(2)*x(5)
↦ j[3]=-x(2)*x(4)+x(1)*x(5)
↦ j[4]=-x(3)^2+x(2)*x(4)
↦ j[5]=-x(2)*x(3)+x(1)*x(4)
↦ j[6]=-x(2)^2+x(1)*x(3)
```

See Section 5.1.17 [det], page 113.

5.1.64 minres

Syntax: `minres` (list)

Type: list

Syntax: `minres` (resolution)

Type: resolution

Purpose: minimizes a free resolution of an ideal or module given by the list resp. resolution argument.

Example:

```
ring r1=32003,(x,y),dp;
ideal i=x5+xy4,x3+x2y+xy2+y3;
resolution rs=lres(i,0);
rs;
↦   1       2       1       0
↦ r1  <-- r1  <-- r1  <-- r1
↦
↦ 0       1       2       3
↦ resolution not minimized yet
↦
list(rs);
↦ [1]:
↦    _[1]=x3+x2y+xy2+y3
↦    _[2]=xy4
↦    _[3]=y7
↦ [2]:
↦    _[1]=-y4*gen(1)+x2*gen(2)+xy*gen(2)+y2*gen(2)+gen(3)
↦    _[2]=-y3*gen(2)+x*gen(3)
```

```
minres(rs);
↦   1       2       1       0
↦ r1  <-- r1  <-- r1  <-- r1
↦
↦ 0       1       2       3
↦
list(rs);
↦ [1]:
↦    _[1]=x3+x2y+xy2+y3
↦    _[2]=xy4
↦ [2]:
↦    _[1]=xy4*gen(1)-x3*gen(2)-x2y*gen(2)-xy2*gen(2)-y3*gen(2)
```

See Section 5.1.96 [res], page 162; Section 5.1.67 [mres], page 144; Section 5.1.104 [sres], page 167.

5.1.65 modulo

Syntax: `modulo ( ideal_expression, ideal_expression )`
`modulo ( module_expression, module_expression )`

Type: module

Purpose: `modulo(h1,h2)` represents $h_1/(h_1 \cap h_2) \cong (h_1+h_2)/h_2)$ where h_1 and h_2 are considered as submodules of the same free module R^l (l=1 for ideals). Let H_1 resp. H_2 be the matrices of size l x k resp. l x m having the columns of h_1 resp. h_2 as generators: $R^k \xrightarrow{H_1} R^l \xleftarrow{H_2} R^m$ Then $h_1/(h_1 \cap h_2) \cong R^k/ker(\overline{H_1})$ where $\overline{H_1} : R^k \to R^l/Im(H_2) = R^l/h_2$ is the induced map.
`modulo(h1,h2)` returns generators of the kernel of this induced map.

Example:

```
ring r;
ideal h1 = x,y,z;
ideal h2 = x;
module m=modulo(h1,h2);
print(m);
↦ 1,0, 0,0,
↦ 0,-z,x,0,
↦ 0,y, 0,x
```

See Section 5.1.111 [syz], page 173.

5.1.66 monitor

Syntax: `monitor ( string_expression )`
`monitor ( string_expression, string_expression )`

Type: none

Purpose: controls recording all user input and/or program output into a file. The second argument describes what to log: `"i"` means input, `"o"` means output, `"io"` for both. The default for the second argument is `"i"`.

Each `monitor` command closes a previous monitor file and opens the file given by the first string expression.
`monitor ("")` turns off recording.

Example:

```
monitor("doe.tmp","io"); // log input and output to doe.tmp
ring r;
poly f=x+y+z;
int i=7;
ideal I=f,x,y;
monitor("");             // stop logging
```

5.1.67 mres

Syntax: `mres ( ideal_expression , int_expression )`
`mres ( module_expression , int_expression )`

Type: resolution

Purpose: computes a minimal free resolution of an ideal or module M with the standard basis method. More precisely, let A=matrix(M), then `mres` computes a free resolution of $M_1 = coker(A) = coker(A_1)$

$$... \longrightarrow F_2 \xrightarrow{A_2} F_1 \xrightarrow{A_1} F_0 \longrightarrow M_1 \longrightarrow 0,$$

where the columns of the matrix A_1 are a minimal set of generators of M if the basering is local or if M is homogeneous. The computation stops after k steps, if the int expression k is not zero, and returns a list of modules M_i = `module`(A_i), i=1..k.
If k=0 then `mres(M,0)` returns a resolution consisting of not more than n 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 of the first szyzgy module of L[1], etc., until L[p+1], such that L[i] $\neq 0$ for i$\leq$p but L[p+1], the first szyzgy module of L[p], is 0 (if the basering is not a qring).

Note: To access the elements of a resolution, it has to be assigned to a list, which also completes computations and may therefore take time.

Example:

```
ring r  = 31991,(t,x,y,z,w),ls;
ideal M = t2x2+tx2y+x2yz,t2y2+ty2z+y2zw,
          t2z2+tz2w+xz2w,t2w2+txw2+xyw2;
resolution L = mres(M,0);
L;
↦  1      4      15      18      7      1      0
↦ r  <-- r  <-- r   <-- r   <-- r  <-- r  <-- r
↦
↦ 0      1      2       3       4      5      6
↦
// projective dimension of M is 4
```

See Section 4.2 [ideal], page 50; Section 5.1.59 [lres], page 139; Section 4.10 [module], page 83; Section 5.1.96 [res], page 162; Section 5.1.104 [sres], page 167.

5.1.68 mstd

Syntax: `mstd ( ideal_expression )`

Type: list

Purpose: returns a list consisting of a standard basis and a minimal set of generators.

Note: the input must be homogeneous and the ring a polynomial ring (global ordering).

Example:

```
ring r=0,(x,y,z,t),dp;
poly f=x3+y4+z6+xyz;
ideal j=jacob(f),f;
j=homog(j,t);j;
↦ j[1]=3x2+yz
↦ j[2]=4y3+xzt
↦ j[3]=6z5+xyt3
↦ j[4]=0
↦ j[5]=z6+y4t2+x3t3+xyzt3
mstd(j);
↦ [1]:
↦    _[1]=3x2+yz
↦    _[2]=4y3+xzt
↦    _[3]=6z5+xyt3
↦    _[4]=xyzt3
↦    _[5]=y2z2t3
↦    _[6]=yz3t4
↦    _[7]=xz3t4
↦    _[8]=yz2t7
↦    _[9]=xz2t7
↦    _[10]=y2zt7
↦    _[11]=xy2t7
↦ [2]:
↦    _[1]=3x2+yz
↦    _[2]=4y3+xzt
↦    _[3]=6z5+xyt3
↦    _[4]=xyzt3
```

See Section 4.2 [ideal], page 50; Section 5.1.106 [std], page 169; Section 5.1.62 [minbase], page 141.

5.1.69 mult

Syntax: `mult ( ideal_expression )`
`mult ( module_expression )`

Type: int

Purpose: computes the degree of the monomial ideal resp. module generated by the leading monomials of the input.
If the input is a standard basis of a homogeneous ideal then it returns the degree of this ideal.

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

Example:

```
ring r= 32003,(x,y),ds;
poly f = (x^3+y^5)^2+x^2*y^7;
ideal i = std(jacob(f));
mult(i);
↦ 46
mult(std(f));
↦ 6
```

See Section 4.2 [ideal], page 50; Section 5.1.106 [std], page 169; Section 5.1.15 [degree], page 112; Section 5.1.118 [vdim], page 176; Section 5.1.19 [dim], page 114.

5.1.70 nameof

Syntax: `nameof ( expression )`

Type: string

Purpose: returns the name of an expression as string.

Example:

```
int i = 9;
string s = nameof(i);
s;
↦ i
nameof(s);
↦ s
nameof(i+1); //returns the empty string:
↦
nameof(basering);
↦ basering
basering;
↦    ? ‘basering‘ is undefined
↦    ? error occurred in Z line 7: ‘basering;‘
ring r;
nameof(basering);
↦ r
```

See Section 5.1.115 [typeof], page 175; Section 5.1.71 [names], page 146; Section 5.1.97 [reservedName], page 163.

5.1.71 names

Syntax: `names ( )`
`names ( ring_name )`

Type: list of strings

Purpose: returns the names of all user defined variables which are ring independent (this includes the names of procedures) or, in the second case, belong to the given ring.

Example:

```
int i = 9;
ring r;
poly f;
names();
↦ [1]:
↦    r
↦ [2]:
↦    i
↦ [3]:
↦    intersect1
↦ [4]:
↦    quotient4
↦ [5]:
↦    quotient5
↦ [6]:
↦    quotient3
↦ [7]:
↦    quotient2
↦ [8]:
↦    quotient1
↦ [9]:
↦    quotient0
↦ [10]:
↦    quot1
↦ [11]:
↦    quot
↦ [12]:
↦    res
↦ [13]:
↦    groebner
↦ [14]:
↦    stdhilb
↦ [15]:
↦    stdfglm
↦ [16]:
↦    LIB
names(r);
↦ [1]:
↦    f
```

See Section 5.1.70 [nameof], page 146; Section 5.1.97 [reservedName], page 163.

5.1.72 ncols

Syntax: `ncols ( matrix_expression )`
`ncols ( intmat_expression )`
`ncols ( ideal_expression )`

Type: int

Purpose: returns the number of columns of a matrix or an intmat
or the number of given generators of the ideal, including zeros.
Note that `size(ideal)` counts the number of generators which are different from zero.
(Use `nrows` to get the number of rows of a given matrix or intmat.)

Example:

```
ring r;
matrix m[5][6];
ncols(m);
↦ 6
ideal i=x,0,y;
ncols(i);
↦ 3
size(i);
↦ 2
```

See Section 4.9 [matrix], page 80; Section 5.1.75 [nrows], page 149; Section 5.1.102 [size], page 166.

5.1.73 npars

Syntax: `npars ( ring_name )`

Type: int

Purpose: returns the number of parameters of a ring.

Example:

```
ring r=(23,t,v),(x,a(1..7)),lp;
// the parameters are t,v
npars(r);
↦ 2
```

See Section 5.1.81 [par], page 154; Section 5.1.83 [parstr], page 155; Section 4.16 [ring], page 95.

5.1.74 nres

Syntax: `nres ( ideal_expression , int_expression )`
`nres ( module_expression , int_expression )`

Type: resolution

Purpose: computes a minimal free resolution of an ideal or module M with the standard basis method. More precisely, let A_1=matrix(M), then `nres` computes a free resolution of $M_1 = coker(A_1)$

$$... \longrightarrow F_2 \xrightarrow{A_2} F_1 \xrightarrow{A_1} F_0 \longrightarrow M_1 \longrightarrow 0,$$

where the columns of the matrix A_1 are the given set of generators of M. The computation stops after k steps, if the int expression k is not zero, and returns a list of modules M_i = `module`(A_i), i=1..k.
If k=0, `nres(M,0)` returns a list of n modules where n is the number of variables of the basering. Let `list l=nres(M,0);` then L[1]=M is identical to the input, L[2] is a minimal set of generators of the first szyzgy module of L[1], etc. (L[i] = M_i in the notations from above).

Example:

```
ring r  = 31991,(t,x,y,z,w),ls;
ideal M = t2x2+tx2y+x2yz,t2y2+ty2z+y2zw,
          t2z2+tz2w+xz2w,t2w2+txw2+xyw2;
resolution L = nres(M,0);
L;
↦  1      4      15      18      7      1      0
↦ r  <-- r  <-- r   <-- r   <-- r  <-- r  <-- r
↦
↦ 0      1      2       3       4      5      6
↦ resolution not minimized yet
↦
```

See Section 4.2 [ideal], page 50; Section 5.1.59 [lres], page 139; Section 4.10 [module], page 83; Section 5.1.67 [mres], page 144; Section 5.1.96 [res], page 162; Section 5.1.104 [sres], page 167.

5.1.75 nrows

Syntax: `nrows ( matrix_expression )`
`nrows ( intmat_expression )`
`nrows ( intvec_expression )`
`nrows ( module_expression )`
`nrows ( vector_expression )`

Type: int

Purpose: returns the number of rows of a matrix, an intmat or an intvec, resp.
the minimal rank of a free module in which the given module or vector lives (the index of the last non-zero component). (Use `ncols` to get the number of columns of a given matrix or intmat.)

Example:

```
ring R;
matrix M[2][3];
nrows(M);
↦ 2
nrows(freemodule(4));
↦ 4
module m=[0,0,1];
nrows(m);
↦ 3
nrows([0,x,0]);
↦ 2
```

See Section 5.1.35 [gen], page 123; Section 4.9 [matrix], page 80; Section 4.10 [module], page 83; Section 4.18 [vector], page 99; Section 5.1.72 [ncols], page 147.

5.1.76 nvars

Syntax: `nvars ( ring_name )`

Type: int

Purpose: returns the number of variables of a ring.

Example:

```
ring r=23,(x,a(1..7)),ls;
nvars(r);
↦ 8
```

See Section 4.16 [ring], page 95; Section 5.1.116 [var], page 176.

5.1.77 open

Syntax: `open ( link_expression );`

Type: none

Purpose: opens a link.

Example:

```
link l="MPtcp:launch";
open(l); // start SINGULAR "server" on localhost in batchmode
close(l); // shut down SINGULAR server
```

See Section 4.6 [link], page 66; Section 5.1.8 [close], page 107.

5.1.78 option

Syntax: `option ();`

Type: string

Purpose: lists all defined options.

Syntax: `option ( option_name );`

Type: none

Purpose: sets an option.

Note: To disable an option, use the prefix `no`.

Syntax: `option ( get );`

Type: intvec

Purpose: dumps the state of all options to an intvec.

Syntax: `option ( set,` intvec_expression `);`

Type: none

Purpose: restores the state of all options from an intvec (produced by `option(get)`).

Values: The following options are used to manipulate the behaviour of computations:

`none` resets all options to the default

`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 find HC (highest corner of the staircase) as fast as possible during a standard basis computation (only used for local orderings)

`intStrategy`
avoids divisions of coefficients during standard basis computations

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

`morePairs`
creates addditional (useless) pairs to speed up computation in some cases

`notRegularity`
disables the regularity bound for res/mres (see Section 5.1.95 [regularity], page 161)

`notSugar` disables the sugar strategy

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

`redSB` computes a reduced standard basis in any standard basis computation

`redTail` reduction of the tails of polynomials during standard basis computations

`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

There are further options that control the behaviour of the computations, but these options are not manipulated using the `option` command:

`multBound`
a multiplicity bound is set (see Section 5.3.4 [multBound], page 186)

`degBound` a degree bound is set (see Section 5.3.1 [degBound], page 184)

The last set of options controls the output of SINGULAR:

`Imap` shows the mapping of variables with the imap command

`loadLib` shows loading of libraries (default)

`debugLib` warns about syntax errors during loading of libraries

`loadProc`	shows loading of procedures from libraries
`mem`	shows memory usage in square brackets (see Section 5.1.61 [memory], page 141)
`prompt`	shows prompt (`>` resp. `.`) if ready for input (default)
`reading`	shows the number of characters read from a file
`redefine`	warns about variable redefinitions (default)
`usage`	shows correct usage in error messages (default)

Example:

```
option(prot);
option();
↦ //options: prot redefine usage prompt
option(notSugar);
option();
↦ //options: prot notSugar redefine usage prompt
option(noprot);
option();
↦ //options: notSugar redefine usage prompt
option(none);
option();
↦ //options: none
ring r=0,x,dp;
degBound=22;
option();
↦ //options: degBound redTail intStrategy
intvec i=option(get);
option(none);
option(set,i);
option();
↦ //options: degBound redTail intStrategy
```

The output reported on `option(prot)` has the following meaning:

`facstd`	`F`	found a new factor all other characters: like the output of `std`
`fglm`	`.`	basis monomial found
	`+`	edge monomial found
	`-`	border monomial found
`groebner`		all characters: like the output of `std`
`lres`	`.`	minimal syzygy found
	n	slanted degree, i.e., row of Betti matrix
	`(m`n`)`	calculate in module n
	`g`	pair found giving reductum and syzygy
`mres`	`[`d`]`	computations of the d-th syzygy module all other characters: like the output of `std`
`minres`	`[`d`]`	minimizing of the d-th syzygy module

`mstd`		all characters: like the output of `std`
`res`	`[`d`]`	computations of the d-th syzygy module all other characters: like the output of `std`
`sres`	`.`	syzygy found
	`(`n`)`	n elements remaining
	`[`n`]`	finished module n
`std`	`s`	found a new element of the standard basis
	`-`	reduced a pair/S-polynomial to 0
	`.`	postponed a reduction of a pair/S-polynomial
	`h`	used Hilbert series criterion
	`H(`d`)`	found a 'highest corner' of degree d, no need to consider higher degrees
	`(`n`)`	n critical pairs are still to be reduced
	`(S:`n`)`	doing complete reduction of n elements
	d	the degree of the leading terms is currently d
`stdfglm`		all characters in first part: like the output of `std` all characters in second part: like the output of `fglm`
`stdhilb`		all characters: like the output of `std`
`syz`		all characters: like the output of `std`

See Section 5.3.1 [degBound], page 184; Section 5.3.4 [multBound], page 186; Section 5.1.106 [std], page 169.

5.1.79 ord

Syntax: `ord ( poly_expression )`
`ord ( vector_expression )`

Type: int

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

Example:

```
ring r=7,(x,y),wp(2,3);
ord(0);
↦ -1
poly f=x2+y3;
ord(f);
↦ 9
ring R=7,(x,y),ws(2,3);
poly f=x2+y3;
ord(f);
↦ 4
vector v=[x2,y];
ord(v);
↦ 3
```

See Section 5.1.14 [deg], page 111; Section 4.12 [poly], page 89; Section 4.18 [vector], page 99.

5.1.80 ordstr

Syntax: `ordstr ( ring_name )`

Type: string

Purpose: returns the description of the monomial ordering of the ring.

Example:

```
ring r=7,(x,y),wp(2,3);
ordstr(r);
↦ wp(2,3),C
```

See Section 4.16 [ring], page 95; Section 5.1.117 [varstr], page 176; Section 5.1.83 [parstr], page 155; Section 5.1.6 [charstr], page 106.

5.1.81 par

Syntax: `par ( int_expression )`

Type: number

Purpose: returns the n-th parameter of the basering. This command should only be used if the basering has at least one parameter.

Example:

```
ring r=(0,a,b,c),(x,y,z),dp;
par(2);
↦ (b)
```

See Section 5.1.73 [npars], page 148; Section 5.1.83 [parstr], page 155; Section 4.16 [ring], page 95; Section 5.1.116 [var], page 176.

5.1.82 pardeg

Syntax: `pardeg( number_expression )`

Type: int

Purpose: returns the degree of a number considered as a polynomial in the ring parameters.

Example:

```
ring r=(0,a,b,c),(x,y,z),dp;
pardeg(a^2*b);
↦ 3
```

See Section 4.11 [number], page 86; Section 4.16 [ring], page 95; Section 5.1.116 [var], page 176.

5.1.83 parstr

Syntax: `parstr ( ring_name )`
`parstr ( int_expression )`
`parstr ( ring_name, int_expression )`

Type: string

Purpose: returns the list of parameters of the ring as a string
or the name of the n-th parameter for an integer n.
`parstr(n)` is equivalent to `parstr(basering,n)`.

Example:

```
ring r=(7,a,b,c),(x,y),wp(2,3);
parstr(r);
↦ a,b,c
parstr(2);
↦ b
parstr(r,3);
↦ c
```

See Section 5.1.6 [charstr], page 106; Section 5.1.73 [npars], page 148; Section 5.1.80 [ordstr], page 154; Section 5.1.81 [par], page 154 Section 4.16 [ring], page 95; Section 5.1.117 [varstr], page 176.

5.1.84 pause

Syntax: `pause;`

Type: none

Purpose: pauses the execution of a procedure until the return key is pressed.

Example:

```
// the procedure will continue if return is pressed:
"press the return key to continue"; pause;
↦ press the return key to continue
↦ pause>
```

5.1.85 preimage

Syntax: `preimage ( ring_name, map_name, ideal_name )`
`preimage ( ring_name, ideal_expression, ideal_name )`

Type: ideal

Purpose: returns the preimage of an ideal under a given map.
The second argument has to be a map from the basering to the given ring (or an ideal defining such a map), and the ideal has to be an ideal in the given ring.

Note: To compute the kernel of a map, the preimage of zero has to be determined. Hence there is no special command for computing the kernel of a map in SINGULAR.

Example:

```
ring r1=32003,(x,y,z,w),lp;
ring r=32003,(x,y,z),dp;
ideal i=x,y,z;
ideal i1=x,y;
ideal i0=0;
map f=r1,i;
setring r1;
ideal i1=preimage(r,f,i1);
i1;
↦ i1[1]=w
↦ i1[2]=y
↦ i1[3]=x
// the kernel of f
preimage(r,f,i0);
↦ _[1]=w
```

See Section 4.8 [map], page 78; Section 4.16 [ring], page 95.

5.1.86 prime

Syntax: `prime ( int_expression )`

Type: int

Purpose: returns the largest prime less then 32004 smaller or equal to the argument; returns 2 for all arguments smaller than 3.

Example:

```
prime(32004);
↦ 32003
prime(0);
↦ 2
prime(-1);
↦ 2
```

See Section 4.3 [int], page 54; Section D.3 [general_lib], page 250.

5.1.87 print

Syntax: `print ( expression );`
`print ( expression, format_string );`

Type: none

Purpose: prints expressions in a nice format, especially useful for matrices, modules and vectors. The second form uses a string to determine the format.
At the moment only the "betti" format is used:

`"betti"` displays the graded Betti numbers of R^n/M, if R denotes the basering and if M is a homogeneous submodule of R^n:

The entry d at (i,j) is the minimal number of generators in degree i+j of the j-th syzygy module of R^n/M (the 0-th (resp.1-st) syzygy module of R^n/M is R^n (resp. M))

Example:

```
ring r=0,(x,y,z),dp;
module m=[1,y],[0,x+z];
m;
↦ m[1]=y*gen(2)+gen(1)
↦ m[2]=x*gen(2)+z*gen(2)
print(m);  // the columns generate m
↦ 1,0,
↦ y,x+z
intmat M=betti(mres(m,0));
print(M,"betti");
↦            0     1
↦ ------------------
↦     0:     2     2
↦ ------------------
↦ total:     2     2
```

See Section 5.1.12 [dbprint], page 110; Section 5.3.7 [short], page 187; Section 5.1.114 [type], page 174; Section 5.1.3 [betti], page 104.

5.1.88 prune

Syntax: `prune ( module_expression )`

Type: module

Purpose: returns the module minimally embedded in a free module such that the corresponding factor modules are isomorphic.

Example:

```
ring r=0,(x,y,z),dp;
module m=gen(1),gen(3),[x,y,0,z],[x+y,0,0,0,1];
print(m);
↦ 1,0,x,x+y,
↦ 0,0,y,0,
↦ 0,1,0,0,
↦ 0,0,z,0,
↦ 0,0,0,1
print(prune(m));
↦ y,
↦ z
```

See Section 4.10 [module], page 83.

5.1.89 qhweight

Syntax: `qhweight ( ideal_expression )`

Type: intvec

Purpose: computes the weight vector of the variables for a quasihomogeneous ideal. If the input is not weighted homogeneous, an intvec of zeroes is returned.

Example:

```
ring h1=32003,(t,x,y,z),dp;
ideal i=x4+y3+z2;
qhweight(i);
↦ 1,3,4,6
```

See Section 4.2 [ideal], page 50; Section 4.5 [intvec], page 63; Section 5.1.120 [weight], page 177.

5.1.90 quote

Syntax: `quote` (expression)

Type: none

Purpose: prevents expressions from evaluation. Used only in connections with write in MPfile links, prevents evalution of an expression before sending to an other SINGULARprocess. Within a quoted expression, the quote can be "undone" by an `eval` (i.e., each eval "undoes" the effect of exactly one `quote`).

Example:

```
link l="MPfile:w example.mp";
ring r=0,(x,y,z),ds;
ideal i=maxideal(3);
ideal j=x7,x2,z;
option(prot);
// compute i+j before writing, but not std
write (l, quote(std(eval(i+j))));
close(l);
// now read it in again and evaluate:
read(l);
↦ 1(12)s2-s3---s-s------7-
↦ product criterion:4 chain criterion:0
↦ _[1]=z
↦ _[2]=x2
↦ _[3]=xy2
↦ _[4]=y3
close(l);
```

See Section 5.1.22 [eval], page 116; Section 5.1.121 [write], page 178; Section 4.6.5.1 [MPfile links], page 69.

5.1.91 quotient

Syntax: `quotient` (ideal_expression, ideal_expression)
`quotient` (module_expression, module_expression)

Type: ideal

Syntax: `quotient ( module_expression, ideal_expression )`

Type: module

Purpose: computes the ideal quotient resp. module quotient.
`quotient(i,j)`$= \{a \in \text{basering} \mid a \cdot j \subset i\}$ in the first case and
`quotient(m,j)`$= \{b \in \text{basering}^n \mid bj \subset m\}$, where $m \subset \text{basering}^n$, in the second case.

Example:

```
ring r=181,(x,y,z),(c,ls);
ideal id1=maxideal(3);
ideal id2=x2+xyz,y2-z3y,z3+y5xz;
ideal id6=quotient(id1,id2);
id6;
↦ id6[1]=z
↦ id6[2]=y
↦ id6[3]=x
quotient(id2,id1);
↦ _[1]=z2
↦ _[2]=yz
↦ _[3]=y2
↦ _[4]=xz
↦ _[5]=xy
↦ _[6]=x2
module m=x*freemodule(3),y*freemodule(2);
ideal id3=x,y;
quotient(m,id3);
↦ _[1]=[1]
↦ _[2]=[0,1]
↦ _[3]=[0,0,x]
```

See Section 4.2 [ideal], page 50; Section 4.10 [module], page 83.

5.1.92 random

Syntax: `random ( min_integer, max_integer )`

Type: int

Purpose: returns random integer between min_integer and max_integer.

Syntax: `random ( max_integer, rows, cols )`

Type: intmat

Purpose: returns a random intmat of size rows x cols with entries between -max_integer and +max_integer (inclusively).

Note: The random generator can be set to a startvalue with the function `system` by a commandline option.

Example:

```
random(1,1000);
↦ 35
random(1,2,3);
↦ 0,0,0,
↦ 1,1,-1
// start the random generator with 210
system("random",210);
random(-1000,1000);
↦ 707
random(-1000,1000);
↦ 284
system("random",210);
random(-1000,1000);    // the same random values again
↦ 707
```

See Section 3.1.6 [Command line options], page 18; Section 4.3 [int], page 54; Section 4.4 [intmat], page 60; Section 5.1.110 [system], page 172.

5.1.93 read

Syntax: `read` (link_expression)
for DBM links:
`read` (link_expression)
`read` (link_expression, string_expression)

Type: any

Purpose: reads data from a link.
For ASCII links, the content of the entire file is returned as one string. If the ASCII link is the empty string, `read` reads from standard input the keyboard.
For MP links, one expression is read from the link, and returned after an evaluation. As long as there is no data to read from an MPtcp link the `read` command blocks (i.e., does not return). The `status` command can be used to check whether or not there is data to 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.

Example:

```
ring r=32003,(x,y,z),dp;
ideal i=x+y,z3+22y;
// write the ideal i to the file save_i
write(":w save_i",i);
ring r0=0,(x,y,z),Dp;
// create an ideal k equal to the content
// of the file save_i
string s="ideal k="+read("save_i")+";";
execute s;
k;
↦ k[1]=x+y
↦ k[2]=z3+22y
```

See Section 5.1.23 [execute], page 117; Section 5.1.36 [getdump], page 124; Section 4.6 [link], page 66; Section 5.1.105 [status], page 168; Section 5.1.121 [write], page 178.

5.1.94 reduce

Syntax: `reduce ( poly_expression, ideal_expression )`
`reduce ( poly_expression, ideal_expression, int_expression )`
`reduce ( vector_expression, ideal_expression )`
`reduce ( vector_expression, ideal_expression, int_expression )`
`reduce ( vector_expression, module_expression )`
`reduce ( vector_expression, module_expression, int_expression )`
`reduce ( ideal_expression, ideal_expression )`
`reduce ( ideal_expression, ideal_expression, int_expression )`
`reduce ( module_expression, ideal_expression )`
`reduce ( module_expression, ideal_expression, int_expression )`
`reduce ( module_expression, module_expression )`
`reduce ( module_expression, module_expression, int_expression )`

Type: the type of the first argument

Purpose: reduces a polynomial, vector, ideal or module to its normal form with respect to an ideal or module represented by a standard basis. Return 0 if and only if the polynomial (resp. vector, ideal, module) is an element (resp. subideal, submodule) of the ideal (resp. module). The result may have no meaning if the second argument is not a standard basis.
The third (optional) argument 1 forces a reduction which considers only the leading term and does no tail reduction.

Note: `NF` may be used instead of `reduce`.

Example:

```
ring r1 = 0,(z,y,x),ds;
poly s1=2x5y+7x2y4+3x2yz3;
poly s2=1x2y2z2+3z8;
poly s3=4xy5+2x2y2z3+11x10;
ideal i=s1,s2,s3;
ideal j=std(i);
reduce(3z3yx2+7y4x2+yx5+z12y2x2,j);
↦ -yx5+2401/81y14x2+2744/81y11x5+392/27y8x8+224/81y5x11+16/81y2x14
reduce(3z3yx2+7y4x2+yx5+z12y2x2,j,1);
↦ -yx5+z12y2x2
```

See Section 4.2 [ideal], page 50; Section 4.18 [vector], page 99; Section 5.1.106 [std], page 169.

5.1.95 regularity

Syntax: `regularity ( list_expression )` `regularity ( resolution_expression )`

Type: int

Purpose: computes the regularity of a homogeneous ideal resp. module from a minimal resolution given by the list expression.

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 the regularity is computed and used as a degree bound during the computation unless `option(notRegularity);` is given.

Example:

```
ring rh3=32003,(w,x,y,z),(dp,C);
poly f= x11+y10+z9+x5*y2+x2*y2*z3+x*y^3*(y2+x)^2;
ideal j= homog(jacob(f),w);
def jr=res(j,0);
regularity(jr);
↦ 23
list jj=jr;
regularity(jj);
↦ 23
```

See Section 5.1.96 [res], page 162; Section 5.1.67 [mres], page 144; Section 5.1.104 [sres], page 167; Section 5.1.64 [minres], page 142; Section 5.1.78 [option], page 150.

5.1.96 res

Syntax: `res ( ideal_expression, int_expression )`
`res ( module_expression, int_expression )`

Type: resolution

Purpose: computes a minimal free resolution of an ideal or module M using a heuristically choosen method. More precisely, let A_1=matrix(M), then `res` computes a free resolution of $M_1 = coker(A_1)$

$$\ldots \longrightarrow F_2 \xrightarrow{A_2} F_1 \xrightarrow{A_1} F_0 \longrightarrow M_1 \longrightarrow 0,$$

where the columns of the matrix A_1 are the given set of generators of M. The computation stops after k steps, if the int expression k is not zero, and returns a list of modules M_i = `module`(A_i), i=1..k.
If k=0, `res(M,0)` returns a list of n modules where n is the number of variables of the basering. Let `list l=res(M,0);` then L[1]=M is identical to the input, L[2] is a minimal set of generators of the first szyzgy module of L[1], etc. (L[i] = M_i in the notations from above).

Note: To access the elements of a resolution, it has to be assigned to a list, which also completes computations and may therefore take time.

Example:

```
ring r  = 31991,(t,x,y,z,w),ls;
ideal M = t2x2+tx2y+x2yz,t2y2+ty2z+y2zw,
          t2z2+tz2w+xz2w,t2w2+txw2+xyw2;
resolution L = res(M,0);
L;
↦  1      4      15      18      7      1      0
```

```
↦ r  <-- r  <-- r   <-- r   <-- r  <-- r  <-- r
↦
↦ 0      1      2       3       4      5      6
↦ resolution not minimized yet
↦
betti(L);
↦ 1,0,0,0,0,0,
↦ 0,0,0,0,0,0,
↦ 0,0,0,0,0,0,
↦ 0,4,0,0,0,0,
↦ 0,0,0,0,0,0,
↦ 0,0,0,0,0,0,
↦ 0,0,6,0,0,0,
↦ 0,0,9,16,2,0,
↦ 0,0,0,2,5,1
```

See Section 4.2 [ideal], page 50; Section 5.1.59 [lres], page 139; Section 4.10 [module], page 83; Section 5.1.67 [mres], page 144; Section 5.1.74 [nres], page 148; Section 5.1.104 [sres], page 167; Section 5.1.111 [syz], page 173.

5.1.97 reservedName

Syntax: `reservedName ()`

Type: none

Syntax: `reservedName (` string_expression `)`

Type: int

Purpose: prints a list of all reserved identifiers (first form) or tests whether the string is a reserved identifier.

Example:

```
reservedName();
↦ ... // output skipped
reservedName("ring");
↦ 1
reservedName("xyz");
↦ 0
```

See Section 5.1.71 [names], page 146.

5.1.98 resultant

Syntax: `resultant (` poly_expression`,` poly_expression`,` ring_variable `)`

Type: poly

Purpose: computes the resultant of the first and second argument with respect to the variable given as the third argument.

Example:

```
ring r=32003,(x,y,z),dp;
poly f=3*(x+2)^3+y;
poly g=x+y+z;
resultant(f,g,x);
↦ 3y3+9y2z+9yz2+3z3-18y2-36yz-18z2+35y+36z-24
```

See Section 4.12 [poly], page 89; Section 4.16 [ring], page 95.

5.1.99 rvar

Syntax: `rvar` (name)
`rvar` (poly_expression)
`rvar` (string_expression)

Type: int

Purpose: returns the number of the variable (hence its boolean value is TRUE) if the name is a ring variable or if the string is the name of a ring variable of the basering.

Example:

```
ring r=29,(x,y,z),lp;
rvar(x);
↦ 1
rvar(r);
↦ 0
rvar(y);
↦ 2
rvar(var(3));
↦ 3
rvar("x");
↦ 1
```

See Section 5.1.13 [defined], page 111; Section 4.16 [ring], page 95; Section 5.1.116 [var], page 176; Section 5.1.117 [varstr], page 176.

5.1.100 setring

Syntax: `setring` ring_name;

Type: none

Purpose: changes the basering to another (already defined) ring.

Example:

```
ring r1=0,(x,y),lp;
// the basering is r1
ring r2=32003,(a(1..8)),ds;
// the basering is r2
setring r1;
// the basering is again r1
nameof(basering);
listvar();
```

Use in procedures:
All changes of the basering by a definition of a new ring or a setring command in a procedure are local to this procedure. Use keepring to move a ring which is local to a procedure up by one nesting level.

See Section 4.14 [qring], page 93; Section 4.16 [ring], page 95; Section 5.2.7 [keepring], page 182.

5.1.101 simplify

Syntax: `simplify ( poly_expression, int_expression )`
`simplify ( vector_expression, int_expression )`
`simplify ( ideal_expression, int_expression )`
`simplify ( module_expression, int_expression )`

Type: the type of the first argument

Purpose: returns the "simplified" first argument depending on the simplification rule given as the second argument.
The simplifications rules are the sum of the following functions:

`1`	normalize (make leading coefficients 1)
`2`	erase zero generators/columns
`4`	keep only the first from identical generators/columns
`8`	keep only the first from generators/columns which differ by a factor from the ground field
`16`	keep only the first from generators/columns whose leading monomials differ
`32`	keep only the first from generators/columns whose leading monomials are not divisible.

Example:

```
ring r = 0, (x,y,z), (c, dp);
ideal i = 0, 2x, 2x, 4x, 3x + y, 5x2;
simplify(i, 1);
↦ _[1]=0
↦ _[2]=x
↦ _[3]=x
↦ _[4]=x
↦ _[5]=x+1/3y
↦ _[6]=x2
simplify(i, 2);
↦ _[1]=2x
↦ _[2]=2x
↦ _[3]=4x
↦ _[4]=3x+y
↦ _[5]=5x2
simplify(i, 4);
↦ _[1]=0
↦ _[2]=2x
↦ _[3]=4x
↦ _[4]=3x+y
```

```
↦ _[5]=5x2
simplify(i, 8);
↦ _[1]=0
↦ _[2]=2x
↦ _[3]=3x+y
↦ _[4]=5x2
simplify(i, 16);
↦ _[1]=0
↦ _[2]=2x
↦ _[3]=5x2
simplify(i, 32);
↦ _[1]=0
↦ _[2]=2x
simplify(i, 32 + 2 + 1);
↦ _[1]=x

matrix A[2][3]=x,0,2x,y,0,2y;
simplify(A,2+8); // by automatic conversion to module
↦ _[1]=[x,y]
```

See Section 4.12 [poly], page 89; Section 4.18 [vector], page 99; Section 4.2 [ideal], page 50; Section 4.10 [module], page 83.

5.1.102 size

Syntax:
`size` (string_expression)
`size` (intvec_expression)
`size` (intmat_expression)
`size` (poly_expression)
`size` (vector_expression)
`size` (ideal_expression)
`size` (module_expression)
`size` (matrix_expression)
`size` (list_expression)
`size` (resolution)

Type: int

Purpose:

ideal or module
: returns the number of (non zero) generators.

string, intvec, list or resolution
: returns the length, i.e. the number of characters, entries or elements.

poly or vector
: returns the number of monomials.

matrix or intmat
: returns the number of entries (rows*columns).

Example:

```
string s="hello";
```

```
size(s);
↦ 5
intvec iv=1,2;
size(iv);
↦ 2
ring r=0,(x,y,z),lp;
poly f=x+y+z;
size(f);
↦ 3
vector v=[x+y,0,1];
size(v);
↦ 3
ideal i=f,y;
size(i);
↦ 2
module m=v,[0,1],[0,0,1],2*v;
size(m);
↦ 4
matrix mm[2][2];
size(mm);
↦ 4
```

See Section 4.17 [string], page 96; Section 4.5 [intvec], page 63; Section 4.4 [intmat], page 60; Section 4.12 [poly], page 89; Section 4.18 [vector], page 99; Section 4.2 [ideal], page 50; Section 4.10 [module], page 83; Section 5.1.72 [ncols], page 147; Section 5.1.75 [nrows], page 149.

5.1.103 sortvec

Syntax: `sortvec` (ideal_expression)
`sortvec` (module_expression)

Type: intvec

Purpose: computes the permutation v: I[i] -> I[v[i]] which orders the ideal resp. module by their initial term, starting with the smallest.

Example:

```
ring r=0,(x,y,z),dp;
ideal I=x,y,z,x3,xz;
sortvec(I);
↦ 3,2,1,5,4
```

See Section D.3 [general_lib], page 250.

5.1.104 sres

Syntax: `sres` (ideal_expression, int_expression)
`sres` (module_expression, int_expression)

Type: resolution

Purpose: computes a free resolution of an ideal or module with Schreyer's method. The ideal resp. module has to be a standard basis. More precisely, let M be given by a standard basis and $A_1 = matrix(M)$. Then `sres` computes a free resolution of $M_1 = coker(A_1)$

$$... \longrightarrow F_2 \xrightarrow{A_2} F_1 \xrightarrow{A_1} F_0 \longrightarrow M_1 \longrightarrow 0,$$

The computation stops after k steps, if the int expression k is not zero, and returns a list of modules (given by standard bases) M_i = `module`(A_i), i=1..k.
If k=0, `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 szyzgy module of L[1], etc. (L[i] = M_i in the notations from above).

Note: To access the elements of a resolution, it has to be assigned to a list, which also completes computations and may therefore take time.

Example:

```
ring r  = 31991,(t,x,y,z,w),ls;
ideal M = t2x2+tx2y+x2yz,t2y2+ty2z+y2zw,
          t2z2+tz2w+xz2w,t2w2+txw2+xyw2;
M       = std(M);
resolution L  = sres(M,0);
L;
↦  1      35      141      209      141      43      4      0
↦ r  <-- r   <-- r    <-- r    <-- r    <-- r   <-- r  <-- r
↦
↦ 0      1       2        3        4        5       6      7
↦ resolution not minimized yet
↦
print(betti(L),"betti");
↦             0     1     2     3     4     5
↦ ------------------------------------------
↦     0:      1     0     0     0     0     0
↦     1:      0     0     0     0     0     0
↦     2:      0     0     0     0     0     0
↦     3:      0     4     0     0     0     0
↦     4:      0     0     0     0     0     0
↦     5:      0     0     0     0     0     0
↦     6:      0     0     6     0     0     0
↦     7:      0     0     9    16     2     0
↦     8:      0     0     0     2     5     1
↦ ------------------------------------------
↦ total:      1     4    15    18     7     1
```

See Section 5.1.3 [betti], page 104; Section 4.2 [ideal], page 50; Section 4.3 [int], page 54; Section 5.1.59 [lres], page 139; Section 5.1.64 [minres], page 142; Section 4.10 [module], page 83; Section 5.1.67 [mres], page 144; Section 5.1.96 [res], page 162; Section 5.1.111 [syz], page 173.

5.1.105 status

Syntax: `status ( link_expression, string_expression )`

Type: string

Syntax: `status ( link_expression, string_expression, string_expression )`
`status ( link_expression, string_expression, string_expression, int_expression )`

Type: int

Purpose: returns the status of the link as asked for by the second argument. If a third argument is given, the result of the comparison to the status string is returned: `(status(l,s1)==s2)` is equivalent to `status(l,s1,s2)`. 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 (see Section A.27 [Parallelization with MPtcp links], page 237 for an example). Note that on some systems, the minimium 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"` returns (depending on the type of the link and its status) `""`,`"w"`,`"a"`,`"r"` or `"rw"`

Example:

```
link l=":w example.txt";
status(l,"write");
↦ not ready
open(l);
status(l,"write","ready");
↦ 1
close(l);
```

See Section 4.6 [link], page 66; Section 5.1.77 [open], page 150; Section 5.1.93 [read], page 160; Section 5.1.121 [write], page 178.

5.1.106 std

Syntax: `std ( ideal_expression)`
`std ( module_expression)`
`std ( ideal_expression, intvec_expression )`
`std ( module_expression, intvec_expression )`

Type: ideal or module

Purpose: returns a standard basis of an ideal or module with respect to the monomial ordering of the basering. A standard basis is a set of generators such that the leading terms generate the leading ideal resp. module.
Use an optional second argument as Hilbert series (result of `hilb(i,1)`, see Section 5.1.39 [hilb], page 125), if the ideal resp. module is homogeneous, (Hilbert driven standard basis computation, Section 5.1.108 [stdhilb], page 171).

Example:

```
ring r = 32003, (x,y,z), ds;
poly s1  = 1x3y2 + 151x5y + 169x2y4 + 151x2yz3 + 186xy6 + 169y9;
poly s2  = 1x2y2z2 + 3z8;
poly s3  = 5x4y2 + 4xy5 + 2x2y2z3 + 1y7 + 11x10;
ideal i = s1, s2, s3;
// compute standard basis j
ideal j = std(i);
```

See Section 4.2 [ideal], page 50; Section 4.16 [ring], page 95; Section 5.1.78 [option], page 150; Section 5.1.26 [facstd], page 118; Section 5.1.68 [mstd], page 145; Section 5.1.107 [stdfglm], page 170; Section 5.1.108 [stdhilb], page 171.

5.1.107 stdfglm

Syntax: `stdfglm` (ideal_expression)

Purpose: returns a standard basis of an ideal in the basering, calculated via the FGLM algorithm from the ordering "dp" (degrevlex) to the ordering of the basering.

Syntax: `stdfglm` (ideal_expression, string_expression)

Purpose: returns a standard basis of an ideal in the basering, calculated via the FGLM algorithm from the ordering given by the second argument to the ordering of the basering.

Note: The first argument has to be a zero-dimensional ideal.

Type: ideal

Example:

```
ring r  = 0,(x,y,z),lp;
ideal i = y3+x2, x2y+x2, x3-x2, z4-x2-y;
ideal i1= stdfglm(i);         //uses fglm from "dp" to "lp"
i1;
↦ i1[1]=z12
↦ i1[2]=yz4-z8
↦ i1[3]=y2+y-z8-z4
↦ i1[4]=xy-xz4-y+z4
↦ i1[5]=x2+y-z4
ideal i2= stdfglm(i,"Dp");    //uses fglm from "Dp" to "lp"
i2;
↦ i2[1]=z12
↦ i2[2]=yz4-z8
↦ i2[3]=y2+y-z8-z4
↦ i2[4]=xy-xz4-y+z4
↦ i2[5]=x2+y-z4
```

See Section 4.2 [ideal], page 50; Section 4.16 [ring], page 95; Section 5.1.78 [option], page 150; Section 5.1.26 [facstd], page 118; Section 5.1.68 [mstd], page 145; Section 5.1.106 [std], page 169; Section 5.1.108 [stdhilb], page 171.

5.1.108 stdhilb

Syntax: `stdhilb ( ideal_expression )`
`stdhilb ( ideal_expression, intvec_expression )`

Type: ideal

Purpose: returns a standard basis of a homogeneous ideal with respect to the monomial ordering of the basering (Hilbert driven standard basis computation). First the Hilbert series is computed (if not given).
Use an optional second argument as Hilbert series (result of `hilb(i,1)`, see Section 5.1.39 [hilb], page 125).

Example:

```
ring  r = 0,(x,y,z),lp;
ideal i = y3+x2, x2y+x2, x3-x2, z4-x2-y;
ideal i1= stdhilb(i); i1;
↦ i1[1]=z12
↦ i1[2]=yz4-z8
↦ i1[3]=y2-yz4+y-z4
↦ i1[4]=y2z4-y2+yz4-y+z4
↦ i1[5]=y3-y+z4
↦ i1[6]=y4+y3
↦ i1[7]=xy-xz4-y3
↦ i1[8]=xy3-y3
↦ i1[9]=x2+y3
↦ i1[10]=x2z4+x2+xy3
↦ i1[11]=x2y+x2
↦ i1[12]=x3-x2
// is in this case equivalent to:
intvec v=1,0,0,-3,0,1,0,3,-1,-1;
ideal i2=stdhilb(i,v);
```

See Section 4.2 [ideal], page 50; Section 5.1.39 [hilb], page 125; Section 4.16 [ring], page 95; Section 5.1.78 [option], page 150; Section 5.1.26 [facstd], page 118; Section 5.1.68 [mstd], page 145; Section 5.1.106 [std], page 169; Section 5.1.107 [stdfglm], page 170.

5.1.109 subst

Syntax: `subst ( poly_expression, ring_variable, monomial )`
`subst ( vector_expression, ring_variable, monomial )`
`subst ( ideal_expression, ring_variable, monomial )`
`subst ( module_expression, ring_variable, monomial )`

Type: poly, vector, ideal or module (the same as the first argument)

Purpose: substitutes a ring variable by a monomial (a polynomial of length <=1).

Example:

```
ring r=0,(x,y,z),dp;
poly f=x2+y2+z2+x+y+z;
subst(f,x,3/2);
↦ y2+z2+y+z+15/4
int a=1;
subst(f,y,a);
↦ x2+z2+x+z+2
subst(f,y,z);
↦ x2+2z2+x+2z
```

See Section 4.12 [poly], page 89; Section 4.18 [vector], page 99; Section 4.2 [ideal], page 50; Section 4.10 [module], page 83.

5.1.110 system

Syntax: `system ( expression_list )`

the first expression must be of type string and selects the desired function.

Type: depends on the desired function, may be none

Purpose: interface to internal data and the operating system.
Not all functions work on every platform.

Functions:

`system("sh",string_expression )`
: shell escape, returns the return code of the shell

`system("pid")`
: returns the process number (for creating unique names)

`system("getenv",string_expression)`
: returns shell environment variable given as the second argument

`system("tty")`
: resets the terminal

`system("version")`
: returns the version number of SINGULAR (type int)

`system("contributors")`
: returns names of people who contributed to the Singular kernel

`system("random",int_expression )`
: resets the random generator to the given value and return it (type int)

`system("random")`
: returns the seed of the random generator (type int)

`system("gen")`
: returns the generating element of the multiplicative group of Z/p\{0} (as int) where p is the characteristic of the basering

`system("nblocks" [, ring])`
: returns the number of blocks of `ring`, or the number of parameters of the current basering, if no second argument is given.

`system("HC")`
: returns the order of the highest corner of the last computation or 0

`system("Singular")`
: returns the absolute (path) name of the running SINGULAR (type string)

`system("long_option_name")`
: returns the value of the (command-line) option `"long_option_name"`; for on/off options, the return value is of type int and either 1 or 0; for all other options, the return type is string.

Example for UNIX:

```
// a listing of the current directory:
system("sh","ls");
// execute a shell, return to SINGULAR with exit:
system("sh","sh");
string unique_name="/tmp/xx"+string(system("pid"));
unique_name;
↦ /tmp/xx4711
system("getenv","PATH");
↦ /bin:/usr/bin:/usr/local/bin
system("Singular");
↦ /usr/local/bin/Singular
// report minimal display time
system("--min-time");
↦ 0.5
```

Example for MSDOS:

```
// a listing of the current directory:
system("sh","dir");
// execute a shell, return to SINGULAR with exit:
system("sh","command");
string unique_name="/tmp/xx"+string(system("pid"));
unique_name;
↦ /tmp/xx0
```

5.1.111 syz

Syntax: `syz ( ideal_expression )`
`syz ( module_expression )`

Type: module

Purpose: computes the first syzygy (i.e. the module of relations of the given generators) of the ideal resp. module.

Example:

```
ring R=0,(x,y),(c,dp);
ideal i=x,y;
syz(i);
↦ _[1]=[y,-x]
```

See Section 4.2 [ideal], page 50; Section 5.1.59 [lres], page 139; Section 4.10 [module], page 83; Section 5.1.67 [mres], page 144; Section 5.1.74 [nres], page 148; Section 5.1.96 [res], page 162; Section 5.1.104 [sres], page 167.

5.1.112 trace

Syntax: `trace` (intmat_expression)
`trace` (matrix_expression)

Type: int resp. poly

Purpose: returns the trace of an intmat resp a matrix.

Example:

```
intmat m[2][2]=1,2,3,4;
print(m);
↦       1     2
↦       3     4
trace(m);
↦ 5
```

See Section 4.4 [intmat], page 60; Section 4.9 [matrix], page 80.

5.1.113 transpose

Syntax: `transpose` (intmat_expression)
`transpose` (matrix_expression)

Type: same type as the argument

Purpose: transposes a matrix.

Example:

```
ring R=0,x,dp;
matrix m[2][3]=1,2,3,4,5,6;
print(m);
↦ 1,2,3,
↦ 4,5,6
print(transpose(m));
↦ 1,4,
↦ 2,5,
↦ 3,6
```

See Section 4.4 [intmat], page 60; Section 4.9 [matrix], page 80.

5.1.114 type

Syntax: `type` (name) ;

Type: none

Purpose: lists the [name, level, type and] value of a variable. To display the value of an expression, it is sufficient to type `expression;`.

Example:

```
int i=3;
i;
↦ 3
type i;
↦ // i                    [0]  int 3
```

See Section 5.1.58 [listvar], page 138; Section 5.1.87 [print], page 156; Chapter 4 [Data types], page 49.

5.1.115 typeof

Syntax: `typeof ( expression )`

Type: string

Purpose: returns the type of an expression as string.

Returns the type of the first list element if the expression is an expression list.

Possible types are: `"ideal"`, `"int"`, `"intmat"`, `"intvec"`, `"list"`, `"map"`, `"matrix"`, `"module"`, `"number"`, `"none"`, `"poly"`, `"proc"`, `"qring"`, `"resolution"`, `"ring"`, `"string"`, `"vector"`.

For internal use only are the types `"package"`, `"?unknown type?"`.

Example:

```
int i = 9;
i;
↦ 9
typeof(_);
↦ int
print(i);
↦ 9
typeof(_);
↦ none
type i;
↦ // i                    [0]  int 9
typeof(_);
↦ string
string s = typeof(i);
s;
↦ int
typeof(s);
↦ string
proc p() {  "hello"; return();}
p();
↦ hello
typeof(_);
↦ none
```

See Chapter 4 [Data types], page 49.

5.1.116 var

Syntax: `var ( int_expression )`

Type: poly

Purpose: returns the n-th ring variable for a given integer n.

Example:

```
ring r=0,(x,y,z),dp;
var(2);
↦ y
```

See Section 4.16 [ring], page 95; Section 4.3 [int], page 54; Section 5.1.76 [nvars], page 150; Section 5.1.117 [varstr], page 176.

5.1.117 varstr

Syntax: `varstr ( ring_name )`
`varstr ( int_expression )`
`varstr ( ring_name, int_expression )`

Type: string

Purpose: returns the list of the names of the ring variables as a string or the name of the n-th ring variable for a given integer n. `varstr(n)` is equivalent to `varstr(basering,n)`.

Example:

```
ring r=0,(x,y,z),dp;
varstr(r);
↦ x,y,z
varstr(r,1);
↦ x
varstr(2);
↦ y
```

See Section 4.16 [ring], page 95; Section 4.3 [int], page 54; Section 5.1.76 [nvars], page 150; Section 5.1.116 [var], page 176; Section 5.1.80 [ordstr], page 154; Section 5.1.6 [charstr], page 106; Section 5.1.83 [parstr], page 155.

5.1.118 vdim

Syntax: `vdim ( ideal_expression )`
`vdim ( module_expression )`

Type: int

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

Example:

```
ring r=0,(x,y),ds;
ideal i=x2+y2,x2-y2;
ideal j=std(i);
vdim(j);
↦ 4
```

See Section 4.2 [ideal], page 50; Section 5.1.106 [std], page 169; Section 5.1.19 [dim], page 114; Section 5.1.15 [degree], page 112; Section 5.1.69 [mult], page 145; Section 5.1.48 [kbase], page 132.

5.1.119 wedge

Syntax: `wedge ( matrix_expression, int_expression )`

Type: matrix

Purpose: computes the n-th exterior power of matrix for a given integer n.

Example:

```
ring r;
matrix m[2][3] = x,y,y,z,z,x;
print(m);
↦ x,y,y,
↦ z,z,x
print(wedge(m,2));
↦ xz-yz,-x2+yz,xy-yz
```

See Section 4.9 [matrix], page 80; Section 5.1.63 [minor], page 141; Section 4.3 [int], page 54.

5.1.120 weight

Syntax: `weight ( ideal_expression )`
`weight ( module_expression )`

Type: intvec

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

Example:

```
ring h1=32003,(t,x,y,z),dp;
ideal i=
9x8+y7t3z4+5x4y2t2+2xy2z3t2,
9y8+7xy6t+2x5y4t2+2x2yz3t2,
9z8+3x2y3z2t4;
intvec e=weight(i);
e;
↦ 5,7,5,7
ring r=32003,(a,b,c,d),wp(e);
```

```
map f=h1,a,b,c,d;
ideal i0=std(f(i));
```

See Section 4.2 [ideal], page 50; Section 4.5 [intvec], page 63; Section 5.1.89 [qhweight], page 157.

5.1.121 write

Syntax: `write ( link_expression, expression_list );`
for DBM links:
`write( link, string_expression, string_expression );`
`write( link, string_expression );`

Type: none

Purpose: writes data to a link.
If the link is of type `ASCII`, all expressions are converted to strings (and separated by a newline character) before they are written. As a consequence, only such values which can be converted to a string can be written to an `ASCII` link.
For 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` blocks (i.e. 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.

Example:

```
//write the lines with the values of the variables f and i
//then the value of m+a into the file "outfile"
write(":w outfile",f,i,m+a);
string filename=":a outfile";
//now append the string "that was f,i,m+a" (without the quotes)
// at the end of the file "outfile"
write(filename,"that was f,i,m+a");
// saving and retrieving data:
ring r=32003,(x,y,z),dp;
ideal i=x+y,z3+22y;
write(":w save_i",i);// this writes x+y,z3+22y to the file save_i

ring r=32003,(x,y,z),dp;
string s=read("save_i");  //creates the string x+y,z3+22y
execute "ideal k="+s+";"; // this defines an ideal k which
                          // is equal to i.
```

See Section 5.1.93 [read], page 160; Section 4.6 [link], page 66; Chapter 4 [Data types], page 49; Section 5.1.90 [quote], page 158; Section 5.1.22 [eval], page 116; Section 5.1.20 [dump], page 114.

5.2 Control structures

A sequence of commands surrounded by curly brackets (`{` and `}`) is a so called block. Blocks are used in SINGULAR in order to define procedures and to collect commands belonging to `if`, `else`, `for` and `while` statements and to the `example` part in libraries. Even if the sequence of statements consists of only a single command it has to be surrounded by curly brackets! Variables which are defined inside a block are not local to that block. Note that there is no ending semicolon at the end of the block.

Example:

```
if ( i>j )
{
  // This is the block
  int temp;
  i=temp;
  i=j;
  j=temp;
  kill temp;
}
```

5.2.1 break

Syntax: `break;`

Purpose: leaves the innermost `for`- or `while`-block.

Example:

```
while (1)
{
  ...
  if ( ... )
  {
    break; // leave the while block
  }
}
```

See Section 5.2 [Control structures], page 179; Section 5.2.10 [while], page 183; Section 5.2.5 [for], page 181.

5.2.2 continue

Syntax: `continue;`

Purpose: skips the rest of this loop und jumps to the beginning of the block. This command is only valid inside a `for` or a `while` construct.

Note: Unlike the C-construct it **does not execute the increment statement**. The command `continue` is mainly for internal use.

Example:

```
for (int i = 1 ; i<=10; i=i+1)
{
   ...
   if (i==3) { i=8;continue; }
     // skip the rest if i is 3 and
     // continue with the next i: 8
   i;
}
↦ 1
↦ 2
↦ 8
↦ 9
↦ 10
```

See Section 5.2 [Control structures], page 179; Section 5.2.5 [for], page 181; Section 5.2.10 [while], page 183.

5.2.3 else

Syntax: `if (` boolean_expression `)` true_block `else` false_block

Purpose: executes false_block if the boolean_expression of the `if` statement is false. This command is only valid in combination with an `if` command.

Example:

```
int i=3;
if (i > 5)
{
  "i is bigger than 5";
}
else
{
  "i is smaller than 6";
}
↦ i is smaller than 6
```

See Section 5.2.6 [if], page 181; Section 4.3.5 [boolean expressions], page 58; Section 5.2 [Control structures], page 179.

5.2.4 export

Syntax: `export` name `;`
`export` list_of_names `;`

Purpose: converts a local variable of a procedure to a global one.

Note: `Objects defined in a ring are not automatically exported` when exporting the ring, (use `keepring` instead).

Example:

```
proc p1
{
  int i,j;
  export i;
  intmat m;
  listvar();
  export m;
}
p1();
↦ // m                    [1]  intmat 1 x 1
↦ // j                    [1]  int 0
↦ // i                    [0]  int 0
listvar();
↦ // m                    [0]  intmat 1 x 1
↦ // i                    [0]  int 0
```

See Section 5.2.7 [keepring], page 182.

5.2.5 for

Syntax: `for` (init_commands; boolean_expression; iterate_commands) block

Purpose: repetitive, conditional execution of a command block.
The command init_command is executed first. Then boolean_expression is evaluated. If its value is TRUE the block is executed, otherwise the `for` statement is complete. After each execution of the block, the command iterate_command is executed and boolean_expression is evaluated.
The command `break;` leaves the innermost `for` construct.

Example:

```
// sum of 1 to 10:
int s=0;
for (int i=1; i<=10; i=i+1)
{
   s=s+i;
}
s;
↦ 55
```

See Section 5.2 [Control structures], page 179; Section 4.3.5 [boolean expressions], page 58; Section 5.2.10 [while], page 183; Section 5.2.6 [if], page 181; Section 5.2.1 [break], page 179; Section 5.2.2 [continue], page 179.

5.2.6 if

Syntax: `if` (boolean_expression) true_block
`if` (boolean_expression) true_block `else` false_block

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

Example:

```
int i = 9;
matrix m [i][i];
if (i > 5 and typeof(m) == "matrix")
{
  m[i][i] = i;
}
```

See Section 5.2.3 [else], page 180; Section 5.2.1 [break], page 179; Section 5.2 [Control structures], page 179; Section 4.3.5 [boolean expressions], page 58.

5.2.7 keepring

Syntax: `keepring` name ;

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

Example:

```
proc P1
{
  ring r=0,x,dp;
  keepring r;
}
proc P2
{
  "inside P2: " + nameof(basering);
  P1();
  "inside P2, after call of P1: " + nameof(basering);
}
ring r1= 0,y,dp;
P2();
↦ inside P2: r1
↦ inside P2, after call of P1: r
"at top level: " + nameof(basering);
↦ at top level: r1
```

5.2.8 quit

Syntax: `quit;`

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

Example: quit;

5.2.9 return

Syntax: `return ( expression_list );`
`return ();`

Type: any

Purpose: returns the result(s) of a procedure.
Can only be used inside a procedure.

Example:

```
proc p2
{
  int i,j;
  for(i=1;i<=10;i++)
  {
    j=j+i;
  }
  return(j);
}
// can also return an expression list, i.e., more than value
proc tworeturn ()
{ return (1,2); }
int i,j = tworeturn();
// return type may even depend on the input
proc type_return (int i)
{
  if (i > 0) {return (i);}
  else {return (list(i));}
}
// then we need def type (or list) to collect value
def t1 = type_return(1);
def t2 = type_return(-1);
```

See Chapter 4 [Data types], page 49; Section 4.13 [proc], page 92.

5.2.10 while

Syntax: `while` (boolean_expression) block

Purpose: repetitive, conditional execution of block.
The command `break` leaves the innermost `while` construct.

Example:

```
int i = 9;
while (i>0)
{
   // ... // do something for i=9, 8, ..., 1
   i = i - 1;
}
while (1)
{
   // ...   // do something forever
   if (i == -5) // but leave the loop if i is -5
```

```
    {
      break;
    }
}
```

See Section 5.2 [Control structures], page 179; Section 4.3.5 [boolean expressions], page 58; Section 5.2.1 [break], page 179.

5.2.11 ~ (breakpoint)

Syntax: `~;`

Purpose: sets a breakpoint. 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 can not exceed 80 characters. SINGULAR proceeds with the execution of the command following `~;` as soon as it receives an empty line.

Example:

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

↦ 3
```

See Section 3.8.2 [Break points], page 47.

5.3 System variables

5.3.1 degBound

Type: int

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

Example:

```
degBound = 7;
option();
↦ //options for 'std'-command: degBound
ideal j=std(i);
degBound;
↦ 7
degBound = 0; //resets degree bound to infinity
```

See Section 5.1.14 [deg], page 111; Section 4.3 [int], page 54; Section 5.1.78 [option], page 150.

5.3.2 echo

Type: int

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

Example:

```
echo = 1;
int i = echo;
↦ int i = echo;
```

See Section 4.3 [int], page 54; refvoice.

5.3.3 minpoly

Type: number

Purpose: describes the coefficient field of the current basering as an algebraic extension with the minimal polynomial equal to `minpoly`. Setting the `minpoly` should be the first command after defining the ring.

Note: The minimal polynomial has to be specified in the syntax of a polynomial. Its variable is not one of the ring variables, but the algebraic element which is being adjoined to the field. Algebraic extension are in SINGULAR only possibel over the rational numbers or Z/p, p prime number.
SINGULAR does not check whether the given polynomial is irreducible! It can be checked in advance with the function `factorize` Section 5.1.27 [factorize], page 119.

Example:

```
//(Q[i]/(i^2+1))[x,y,z]:
ring Cxyz=(0,i),(x,y,z),dp;
minpoly=i^2+1;
i2;
↦ (-1)
```

See Section 4.16 [ring], page 95.

5.3.4 multBound

Type: int

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

Example:

```
multBound = 20;
option();
↦    //options for 'std'-command: multBound
ideal j=std(i);
multBound;
↦ 20
multBound = 0;  //disables multBound
```

See Section 4.3 [int], page 54; Section 5.1.69 [mult], page 145; Section 5.1.78 [option], page 150.

5.3.5 noether

Type: poly

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

Example:

```
ring R=32003,(x,y,z),ds;
ideal i=x2+y^12,y13;
std(i);
↦ _[1]=x2+y12
↦ _[2]=y13
noether = x^11;
std(i);
↦ _[1]=x2
noether = 0; //disables noether
```

See Section 4.12 [poly], page 89; Section 5.1.106 [std], page 169.

5.3.6 printlevel

Type: int

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

Example:

```
voice;
↦ 1
```

```
printlevel=0;
dbprint(1);
printlevel=voice;
dbprint(1);
↦ 1
```

See Section 4.3 [int], page 54; Section 5.1.12 [dbprint], page 110; Section 5.3.11 [voice], page 189.

5.3.7 short

Type: int

Purpose: the output of monomials is done in the short manner, if `short` is nonzero. A C-like notion is used, if short is zero. Both notations may be used as input.
The default depends on the names of the ring variables (0 if there are names of variables longer than 1 character, 1 otherwise). Every change of the basering sets `short` to the proper default value for that ring and overwrites the previous setting.

Example:

```
ring r=23,x,dp;
int save=short;
short = 1;
2x2,x2;
↦ 2x2 x2
short = 0;
2x2,x2;
↦ 2*x^2 x^2
short=save;//resets short to the previous value
```

5.3.8 timer

Type: int

Purpose:

1. the CPU time used for each command is printed if timer >0 and if this time is bigger than a (customizable) minimal time;
2. yields the used CPU time since the start-up of SINGULAR in a (customizable) resolution.

The default setting of `timer` is 0, the default minimal time is 0.5 seconds, and the default timer resolution is 1 (i.e., default unit of time is one second). The minimal time and timer resolution can be set using the command line options `--min-time` and `--ticks-per-sec` and can be checked using `system("--min-time")` and `system("--ticks-per-sec")`.

How to use `timer` in order to measure the time for a sequence of commands, see example below.

Example:

```
timer = 1; // The time of each command is printed
int t=timer; // initialize t by timer
ring r=0,(x,y,z),dp;
```

```
poly p=(x+2y+3z+4xy+5xz+6yz)^20;
↦ //used time: 1.55 sec
// timer as int_expression:
t = timer - t;
t;  // yields the time in ticks-per-sec (default 1)
↦ 2
    // since t was initialized by timer
execute "int tps=" + system("--ticks-per-sec");
t/tps; // yields the time in seconds
↦ 2
```

See Section 3.1.6 [Command line options], page 18; Section 5.1.110 [system], page 172; Section 5.3.10 [rtimer], page 188.

5.3.9 TRACE

Type: int

Purpose: sets level of debugging.

`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.
`TRACE` does not affect the output of commands.

Example:

```
TRACE = 1;
LIB "general.lib";
sum(1..100);
↦ entering sum (level 0)
↦ leaving  sum (level 0)
↦ 5050
```

See Section 4.3 [int], page 54.

5.3.10 rtimer

Type: int

Purpose: identical to `timer`, except that real times (i.e., wall-clock) times are reported, instead of CPU times.

Example:

See Section 5.3.8 [timer], page 187; Section 3.1.6 [Command line options], page 18; Section 5.1.110 [system], page 172.

5.3.11 voice

Type: int

Purpose: shows the nesting level of procedures.

Example:

```
voice;
↦ 1
proc p
{
  voice;
}
↦ 2
```

See Section 5.1.12 [dbprint], page 110; Section 5.1.58 [listvar], page 138; Section 5.3.6 [printlevel], page 186.

6 Tricks and pitfalls

6.1 Limitations

SINGULAR has the following limitations:

- the characteristic of a prime field must be less than 32004
- the (weighted) degree of a monomial must be smaller than 2147483648
- the exponent of a ring variable must be smaller than 32768
- a ring must have 505 variables or less
  (501 on a DEC Alpha)
- integers (of type `int`) have the limited range from -2147483647 to 2147483647
- the length of an identifier is unlimited but `listvar` displays only the first 20 characters

6.2 Major differences to the C programming language

Although many constructs from SINGULAR's programming language are similar to those from the C programming language, there are some subtle differences. Most notably:

No rvalue of increments and assignments

The increment `++` (resp. decrement operator `--`) has no rvalue, i.e., cannot be used on the right-hand sides of assignments. So, instead of

```
j = i++;  // WRONG!!!
```

(which results in an error), it must be written

```
i++; j = i;
```

Likewise, an assignment expression does not have a result. Therefore, compound assignments like `i = j = k;` are not allowed and result in an error.

Evaluation of logical expressions

All arguments of a logical expressions are first evaluated and then the value of the logical expression is determined. For example, the logical expressions `(a || b)` is evaluated by first evaluating `a` *and* `b`, even though the value of `b` has no influence on the value of `(a || b)`, if `a` evaluates to true. Therefore, the following results in a syntax error

```
if (defined(i) && i > 0) {} // WRONG!!!
```

if the variable `i` is undefined. This must be written instead as

```
if (defined(i))
{
  if (i > 0) {}
}
```

Note that this evaluation is different from the left-to-right, conditional evaluation of logical expressions (as found in most programming languages). For example, in these other languages, the value of `(1 || b)` is determined without ever evaluating `b`.

However, there are several short work-arounds for this problem:

1. If a variable (say, `i`) is only to be used as a boolean flag, then define (value is true) and undefine (value is false) `i` instead of assigning a value. Using this scheme, it is sufficient to simply write

   ```
   if (defined(i))
   ```

   in order to check whether `i` is true. Use the command `kill` to undefine a variable (see Section 5.1.49 [kill], page 133).
2. If a variable can have more than two values, then define it, if necessary, before it is used for the first time. For example, if the following is used within a procedure

   ```
   if (! defined(DEBUG)) { int DEBUG = 1;}
   ...
   if (DEBUG == 3)  {...}
   if (DEBUG == 2)  {...}
   ...
   ```

   then a user of this procedure does not need to care about the existence of the `DEBUG` variable – this remains hidden from the user. However, if `DEBUG` exists globally, then its local default value is overwritten by its global one.

No case or switch statement

SINGULAR does not offer a `case` (or, `switch`) statement. However, it can be imitated in the following way:

```
while (1)
{
   if (choice == choice_1) { ...; break;}
   ...
   if (choice == choice_n) { ...; break;}
   // default case
   ...; break;
}
```

Usage of commas

In SINGULAR, a comma separates list elements and the value of a comma expression is a list. Hence, commas can not be used to combine several expressions into a single expression. For example, instead of writing

```
for (i=1, j=5; i<5 || j<10; i++, j++) {...} // WRONG!!!!!!
```

one has to write

```
for (i,j = 1,5; i<5 || j<10; i++, j++) {...}
```

Usage of brackets

In SINGULAR, curly brackets (`{ }`) **must always** be used to enclose the statement body following such constructs like `if`, `else`, `for`, `while`, even if this block consists of only a single statement. Similarly, in the return statement of a procedure parentheses (`( )`) **must always** be used to enclose the return value. Even if there is no value to return, parentheses have to be used after a return statement (i.e., `return();`). For example,

```
if (i == 1) return i;    // WRONG!!!!!
```

results in a errors. Instead, it must be written as

```
if (i == 1) { return (i); }
```

Behaviour of continue

SINGULAR's `continue` construct is only valid inside the body of a `for` or `while` construct. It skips the rest of the loop-body and jumps to the beginning of the block. Unlike the C-construct SINGULAR's `continue` **does not execute the increment statement**. For example,

```
for (int i = 1 ; i<=10; i=i+1)
{
   ...
   if (i==3) { i=8;continue; }
     // skip the rest if i is 3 and
     // continue with the next i: 8
   i;
}
↦ 1
↦ 2
↦ 8
↦ 9
↦ 10
```

Although the SINGULAR language is a strongly typed programming language, the type of the return value of a procedure does not need to be specified. As a consequence, the return type of a procedure may vary, i.e., may, for example, depend on the input. However, the return value of such a procedures may then only be assigned to a variable of type `def`.

```
proc type_return (int i)
{
```

```
if (i > 0) {return (i);}
else {return (list(i));}
}
def t1 = type_return(1);
def t2 = type_return(-1);
typeof(t1); typeof(t2);
↦ int
↦ list
```

Furthermore, it is mandatory to assign the return value of a procedure to a variable of type def, if a procedure changes the current ring using the `keepring` command (see Section 5.2.7 [keepring], page 182) and returns a ring-dependent value (like a polynomial or module).

```
proc def_return
{
ring r=0,(x,y),dp;
poly p = x;
keepring r;
return (x);
}
def p = def_return();
// poly p = def_return(); would be WRONG!!!
typeof(p);
↦ poly
```

On the other hand, more than one value can be returned by a single `return` statement. For example,

```
proc tworeturn () { return (1,2); }
int i,j = tworeturn();
```

6.3 Miscellaneous oddities

1. integer division

   A sequence of digits and `/` without spaces is of type number. With spaces it is an expression of type int (and `/` is the integer division). To avoid confusion use the operand `div`.

   ```
   ring r=32002,x,dp;
   3/2;
   ↦ -15994
   3 / 2;
   ↦ 1
   3 div 2;
   ↦ 1
   number(3) / number(2);
   ↦ -15994
   number a=3;
   number b=2;
   a / b;
   ↦ -15994
   ```

```
a div b;
↦    ? `number` div `number` is not supported
↦    ? error occurred in Z line 9: `a div b;`
```

2. monomials and precedence

   The computation of a monomial has precedence over all operators:

```
ring r=0,(x,y),dp;
2xy^2 == (2*x*y)^2;
↦ 1
2xy^2 == 2x*y^2;
↦ 0
2x*y^2 == 2*x * (y^2);
↦ 1
```

3. meaning of mult

   For an arbitrary ideal or module i, mult(i) returns the multiplicity of the ideal generated by the leading monomials of the given generators of i, hence depends on the monomial ordering!

   A standard mistake is to interpret `degree(i)` or `mult(i)` for an inhomogeneous ideal `i` as the degree of the homogenization or as something like the 'degree of the affine part'. For the ordering dp (degree reverse lexicographical) the converse is true: if `i` is given by a standard basis, `mult(i)` is the degree of the homogeneous ideal obtained by homogenization of `i` and then putting the homogenizing variable to 0, hence it is the degree of the part at infinity (this can also be checked by looking at the initial ideal).

4. size of ideals

   `size` counts the non-zero entries of an ideal (resp. module). Use `ncols` to determine the actual number of entries in the ideal (resp. module).

5. computations in qrings

   SINGULAR computes in a quotient rings as long as possible with the given representative of a polynomial, say, `f`. I.e., it usually does not reduce `f` w.r.t. the quotient ideal. This is only done when necessary during standard bases computations or by an explicit reduction using the command `reduce(f, std(0))` (see Section 5.1.94 [reduce], page 161).

6. substring selection

   Two comma – separated `int`s in square brackets after a string select a substring. An `intvec` in square brackets after a string selects the respective single characters of the string and returns them as an expression list of two strings. In particular,

```
string s = "one-word";
s[2,6];     // a substring starting at the second char
↦ ne-wor
size(_);
↦ 6
intvec v = 2,6;
s[v];      // the second and the sixth char
↦ n o
string st = _;  // stick toghter by an assignment
st;
↦ n
size(_);
↦ 1
v = 2,6,8;
s[v];
↦ n o d
```

6.4 Identifier resolution

In SINGULAR, an identifier (i.e., a "word") is resolved in the following way and order: It is checked for

1. a reserved name,
2. a local variable (w.r.t. a procedure),
3. a local ring variable (w.r.t. a ring locally set in a procedure),
4. a global variable,
5. a monomial consisting of local ring variables written without operators,
6. a monomial consisting of global ring variables written without operators.

Consequently, it is allowed to have general variables with the same name as ring variables. However, the above identifier resolution order must be kept in mind. Otherwise, surprising results may come up.

```
ring r=0,(x,y),dp;
int x;
x*y; // resolved product int*poly, i.e. 0*y
↦ 0
xy; // "xy" is one identifier and resolved to monomial xy
↦ xy
```

For these reasons, we strongly recommend not to use variables which have the same name(s) as ring variables.

Appendix A Examples

A.1 Milnor and Tjurina

The Milnor number resp. the Tjurina number of a power series f in $K[[x_1,\ldots,x_n]]$ is

$$milnor(f) = \dim_K(K[[x_1,\ldots,x_n]]/jacob(f))$$

resp.

$$tjurina(f) = \dim_K(K[[x_1,\ldots,x_n]]/((f)+jacob(f)))$$

where jacob(f) is the ideal generated by the partials of f. tjurina(f) is finite, if and only if f has an isolated singularity. The same holds for milnor(f) if K has characteristic 0. SINGULAR displays -1 if the dimension is infinite.

SINGULAR cannot compute with infinite power series. But it can work in $\mathrm{Loc}_{(x)}K[x_1,\ldots,x_n]$, the localization of $K[x_1,\ldots,x_n]$ at the maximal ideal $(x_1,\ldots,x_n)$. To do this one has to define an s-ordering like ds, Ds, ls, ws, Ws or an appropriate matrix ordering (look at the manual to get information about the possible monomial orderings in SINGULAR, or type `help Monomial orderings;` to get a menu of possible orderings. For further help type e.g. `help local orderings;`).

We shall show in the example below how to realize the following:

- set option `prot` to have a short protocol during standard basis computation
- define the ring r1 with char 32003, variables x,y,z, monomial ordering ds, series ring (i.e. K[x,y,z] localized at (x,y,z))
- list the information about r1 by typing its name
- define the integers a,b,c,t
- define a polynomial f (depending on a,b,c,t) and display it
- define the jacobian ideal i of f
- compute a standard basis of i
- compute the Milnor number (=250) with vdim and create and display a string in order to comment the result (text between quotes " "; is a 'string')
- compute a standard basis of i+(f)
- compute the Tjurina number (=195) with `vdim`
- then compute the Milnor number (=248) and the Tjurina number (=195) for t=1
- reset the option to `noprot`

```
option(prot);
ring r1 = 32003,(x,y,z),ds;
r1;
↦ //   characteristic : 32003
↦ //   number of vars : 3
↦ //        block   1 : ordering ds
↦ //                  : names    x y z
↦ //        block   2 : ordering C
int a,b,c,t=11,5,3,0;
poly f = x^a+y^b+z^(3*c)+x^(c+2)*y^(c-1)+x^(c-1)*y^(c-1)*z3+
```

```
        x^(c-2)*y^c*(y^2+t*x)^2;
f;
↦ y5+x5y2+x2y2z3+xy7+z9+x11
ideal i=jacob(f);
i;
↦ i[1]=5x4y2+2xy2z3+y7+11x10
↦ i[2]=5y4+2x5y+2x2yz3+7xy6
↦ i[3]=3x2y2z2+9z8
ideal j=std(i);
↦ 7(2)s8-s10.s11(2)s12.s(3)s13(4)s(5)s14(6)s(7)15--.s(6)-16.-.s(5)17.s(
↦ 7)s.--s18(6).--19-..sH(24)20....21.---
↦ product criterion:10 chain criterion:69
"The Milnor number of f(11,5,3) for t=0 is", vdim(j);
↦ The Milnor number of f(11,5,3) for t=0 is 250
j=i+f;    // overwrite j
j=std(j);
↦ 7(3)s8-s10.s11(3)s.s12(4)s(5)s13(6)s(8)s14(9).s(10).15--sH(23)....16.
↦ .....17.......sH(21)(9)sH(20)16(10).17.---------sH(19)16-
↦ product criterion:10 chain criterion:53
vdim(j);  // compute the Tjurina number for t=0
↦ 195
t=1;
f=x^a+y^b+z^(3*c)+x^(c+2)*y^(c-1)+x^(c-1)*y^(c-1)*z3
  +x^(c-2)*y^c*(y^2+t*x)^2;
ideal i1=jacob(f);
ideal j1=std(i1);
↦ 7(2)s8-s10.s11.s12(2)s13(3)s.s(4)s14(5)s(6)s15(7).....s(8)16.s....s(9
↦ )..17............s18(10).....s(11)..-.19.......sH(24)......20........
↦ ...21.--.......--------
↦ product criterion:11 chain criterion:83
"The Milnor number of f(11,5,3) for t=1:",vdim(j1);
↦ The Milnor number of f(11,5,3) for t=1: 248
vdim(std(j1+f));   // compute the Tjurina number for t=1
↦ 7(16)s8-s10.s11.s.s(16)-12.s(16)-s13(16)s(17)s(18)s(19)-s..-14-s(17)-
↦ s.s.s(17)s15(18)..-s(18)...--.16....-.......s(16).sH(23).s(18)...17..
↦ ........18.........sH(20)17-.-------...s(12).---.----...sH(19)16-----
↦
↦ product criterion:15 chain criterion:174
↦ 195
option(noprot);
```

A.2 Procedures and LIB

The computation of the Milnor number (for an arbitrary isolated complete intersection singularity ICIS) and the Tjurina number (for an arbitrary isolated singularity) can be done by using procedures from the library `sing.lib`. For a hypersurface singularity it is very easy to write a procedure which computes the Milnor number and the Tjurina number.

We shall demonstrate:

- loading the library `sing.lib`
- define a local ring in 2 variables and characteristic 0
- define a plane curve singularity
- compute Milnor number and Tjurina number by using the procedures `milnor` and `tjurina`
- write your own procedures: (A procedure has a list of input parameters and of return values, both lists may be empty.)
  - the procedure `mil` which must be called with one parameter, a polynomial. The name g is local to the procedure and is killed automatically. `mil` returns the Milnor number (and displays a comment).
  - the procedure `tjur` where the parameters are not specified. They are referred to by `#[1]` for the 1-st, `#[2]` for the 2-nd parameter etc. `tjur` returns the Tjurina number (and displays a comment).
  - the procedure `milrina` which returns a list consisting of two integers, the Milnor and the Tjurina number.

```
LIB "sing.lib";
// you should get the information that sing.lib has been loaded
// together with some other libraries which are needed by sing.lib
ring r = 0,(x,y),ds;
poly f = x7+y7+(x-y)^2*x2y2;
milnor(f);
↦ 28
tjurina(f);
↦ 24

proc mil (poly g)
{
   "Milnor number:";
   return(vdim(std(jacob(g))));
}
mil(f);
↦ Milnor number:
↦ 28

proc tjur
{
   "Tjurina number:";
   return(vdim(std(jacob(#[1])+#[1])));
}
tjur(f);
↦ Tjurina number:
↦ 24

proc milrina (poly f)
{
   ideal j=jacob(f);
   list L=vdim(std(j)),vdim(std(j+f));
   return(L);
}
milrina(f);     // a list containing Milnor and Tjurina number
```

```
↦ [1]:
↦    28
↦ [2]:
↦    24
milrina(f)[2];  // the second element of the list
↦ 24
```

A.3 Critical points

The same computation which computes the Milnor resp. the Tjurina number, but with ordering dp instead of ds (i.e. in $K[x_1,\ldots,x_n]$ instead of $\mathrm{Loc}_{(x)}K[x_1,\ldots,x_n]$) gives:

- the number of critical points of f in the affine plane (counted with multiplicities)
- the number of singular points of f on the affine plane curve f=0 (counted with multiplicities)

We start with the ring r1 from section Section A.1 [Milnor and Tjurina], page 196 and its elements.

The following will be realized below:

- reset the protocol option and activate the timer
- define the ring r2 with char 32003, variables x,y,z and monomial ordering dp (= degrevlex) (i.e. the polynomial ring = K[x,y,z]).
- Note that polynomials, ideals, matrices (of polys), vectors, modules belong to a ring, hence we have to define f and jacob(f) again in r2. Since these objects are local to a ring, we may use the same names. Instead of defining f again we map it from ring r1 to r2 by using the `imap` command (`imap` is a convenient way to map variables from some ring identically to variables with the same name in the basering, even if the groundfield is different. Compare with `fetch` which works for almost identical rings, e.g. if the rings differ only by the ordering or the names of the variables and which may be used to rename variables). Integers and strings however do not belong to any ring. Once defined they are globally known.
- The result of the computation here (together with the previous one in Section A.1 [Milnor and Tjurina], page 196) shows that (for t=0) $\dim_K(\mathrm{Loc}_{(x)}K[x_1,\ldots,x_n]/jacob(f)) = 250$ (previously computed) while $\dim_K(K[x_1,\ldots,x_n]/jacob(f)) = 536$. Hence f has 286 critical points, counted with multiplicity, outside the origin. Moreover, since $\dim_K(\mathrm{Loc}_{(x)}K[x_1,\ldots,x_n]/(jacob(f)+(f))) = 195 = \dim_K(K[x_1,\ldots,x_n]/(jacob(f)+(f)))$, the affine surface f=0 is smooth outside the origin.

```
ring r1 = 32003,(x,y,z),ds;
int a,b,c,t=11,5,3,0;
poly f = x^a+y^b+z^(3*c)+x^(c+2)*y^(c-1)+x^(c-1)*y^(c-1)*z3+
         x^(c-2)*y^c*(y^2+t*x)^2;
timer=1;
ring r2 = 32003,(x,y,z),dp;
poly f=imap(r1,f);
ideal j=jacob(f);
vdim(std(j));
↦ 536
vdim(std(j+f));
```

```
↦ 195
timer=0;  // reset timer
```

A.4 Saturation

Since in the example above, the ideal j+(f) has the same vdim in the polynomial ring and in the localization at 0 (each 195), f=0 is smooth outside 0! Hence j+(f) contains some power of the maximal ideal m. We shall check this in a different manner: For any two ideals i, j in the basering R let

$$\begin{aligned}\mathrm{sat}(i,j) &= \{x \in R \mid \exists\, n \text{ s.t. } x * (j^n) \subseteq i\} \\ &= \bigcup_{n=1}^{\infty} i : j^n\end{aligned}$$

denote the saturation of i with respect to j. This defines, geometrically, the closure of the complement of V(j) in V(i) (V(i) denotes the variety defined by i). In our case, sat(j+(f),m) must be the whole ring, hence generated by 1.

The saturation is computed by the procedure `sat` in `elim.lib` by computing iterated ideal quotients with the maximal ideal. `sat` returns a list of two elements: the saturated ideal and the number of iterations. (Note that `maxideal(n)` denotes the n-th power of the maximal ideal).

```
LIB "elim.lib";         // loading library elim.lib
// you should get the information that elim.lib has been loaded
// together with some other libraries which are needed by it
option(noprot);         // no protocol
ring r2 = 32003,(x,y,z),dp;
poly f = x^11+y^5+z^(3*3)+x^(3+2)*y^(3-1)+x^(3-1)*y^(3-1)*z3+
  x^(3-2)*y^3*(y^2)^2;
ideal j=jacob(f);
sat(j+f,maxideal(1));
↦ [1]:
↦    _[1]=1
↦ [2]:
↦    17
// list the variables defined so far:
listvar();
↦ // r2                  [0]  *ring
↦ //      j                   [0]  ideal, 3 generator(s)
↦ //      f                   [0]  poly
↦ // LIB                 [0]  string standard.lib,elim.li..., 52 char(
↦ s)
```

A.5 Long coefficients

The following innocent example produces in its standard basis extremely long coefficients in char 0 for the lexicographical ordering. But a very small deformation does not (the undeformed example is degenerate with respect to the Newton boundary). This example demonstrates that it might be wise, for complicated examples, to do the calculation first in positive char (e.g. 32003). It

was shown, that in complicated examples, more than 95 percent of the time needed for a standard basis computation is used in the computation of the coefficients (in char 0). The representation of long integers with real is demonstrated.

```
timer = 1;                                  // activate the timer
option(prot);
ring R0 = 0,(x,y),lp;
poly f = x5+y11+xy9+x3y9;
ideal i = jacob(f);
ideal i1 = i,i[1]*i[2];                     // undeformed ideal
ideal i2 = i,i[1]*i[2]+1/1000000*x5y8;  // deformation of i1
i1; i2;
↦ i1[1]=5x4+3x2y9+y9
↦ i1[2]=9x3y8+9xy8+11y10
↦ i1[3]=45x7y8+27x5y17+45x5y8+55x4y10+36x3y17+33x2y19+9xy17+11y19
↦ i2[1]=5x4+3x2y9+y9
↦ i2[2]=9x3y8+9xy8+11y10
↦ i2[3]=45x7y8+27x5y17+45000001/1000000x5y8+55x4y10+36x3y17
↦ +33x2y19+9xy17+11y19
ideal j = std(i1);
↦ V19-s20.s21(1)s22(2)23-s27.s28.s29.s30.s31.s32.s33.s34.s35.s36
↦ .s37.s38.s39.s40.s.70-
↦ product criterion:1 chain criterion:30
↦ //used time: 1.9 sec
j;
↦ j[1]=264627y39+26244y35-1323135y30-131220y26+1715175y21
↦ +164025y17+1830125y16
↦ j[2]=-121039477919718467198383218863933929137500650608 75xy8
↦ +2863915211416831987013319392500032667677386328 75y38
↦ +3195440220690902692676462287757356578554430672591y37
↦ -5743662142082266384972138126573889528284632 0y36
↦ -1657764214948799497573918210031067353932439400y35
↦ -2130184815893081911956772238989868269700120550 0y34
↦ -1822194158663066565585991976961565719648069806148y33
↦ +4701709279892816135156972313196394005220175y32
↦ +1351872269688192267600786976008506868242319 75y31
↦ +3873063305929810816961516976025038053001141375y30
↦ -1325886675843874047990382005421144061861290080000y29
↦ -1597720195476063141946794589554240608952696688 7310y28
↦ +2627018133630909266063334800262533042671265 25y27
↦ +7586082690893335269027136248944859544727953125y26
↦ +8678530741064946460228584335167214896539594562 5y25
↦ +5545808143273594102173252331151835700278639247 45y24
↦ -1907556301346043736467915377903839489563832 5y23
↦ -5485623227155017610583489967769225610740211 25y22
↦ -1574654526776483860739574647156810078093398312 5y21
↦ +1414279129721176222978654235817359505555191156250y20
↦ +2071119006944589361521339965003571537816994342 3125y19
↦ -2729427333374726655734180929779053229840097 50y18
↦ -7890651158453345058018472946774133657209553750y17
↦ -6355489703849168678772965606104472465108980 3125y16
```

```
↦ +220992517299239066997322447610282660743502559616 25y14
↦ -14793713967965590435357948972258591339027857296625y10
↦ j[3]=5x4+3x2y9+y9

// Compute average coefficient length (=51) by
//   - converting j[2] to a string in order to compute the number
//   of characters
//   - divide this by the number of monomials:
size(string(j[2]))/size(j[2]);
↦ 51
vdim(j);
↦ 63

// For a better representation normalize the long coefficients
// of the polynomial j[2] and map it  to real:
poly p=-(1/12103947791971846719838321886393392913750065060875)*j[2];
ring R1=real,(x,y),lp;
poly p=imap(R0,p);
p;
↦ -xy8
↦ +2.366e-02y38
↦ +2.640e-01y37
↦ -4.745e-06y36
↦ -1.370e-04y35
↦ -1.760e-03y34
↦ -1.505e-01y33
↦ +3.884e-07y32
↦ +1.117e-05y31
↦ +3.200e-04y30
↦ -1.095e-01y29
↦ -1.320e+00y28
↦ +2.170e-05y27
↦ +6.267e-04y26
↦ +7.170e-03y25
↦ +4.582e-01y24
↦ +1.576e-06y23
↦ -4.532e-05y22
↦ -1.301e-03y21
↦ +1.168e-01y20
↦ +1.711e+00y19
↦ +2.255e-05y18
↦ -6.519e-04y17
↦ -5.251e-03y16
↦ +1.826e+00y14
↦ -1.222e+00y10

// Compute a standard basis for the deformed ideal:
setring R0;
j = std(i2);
↦ v13.s14(2)s15.s16.19-s.s20.s21.23-
↦ product criterion:3 chain criterion:7
```

```
j;
↦ j[1]=y16
↦ j[2]=-729xy8+1331y14-891y10
↦ j[3]=5x4+3x2y9+y9
vdim(j);
↦ 40
```

A.6 Parameters

Let us now deform the above ideal by introducing a parameter t and compute over the ground field Q(t). We compute the dimension at the generic point, i.e. $dim_{Q(t)}Q(t)[x,y]/j$. (This gives the same result as for the deformed ideal above. Hence, the above small deformation was "generic".)

For almost all $a \in Q$ this is the same as $dim_Q Q[x,y]/j_0$, where $j_0 = j|_{t=a}$.

```
ring Rt = (0,t),(x,y),lp;
Rt;
↦ //   characteristic : 0
↦ //   1 parameter    : t
↦ //   minpoly        : 0
↦ //   number of vars : 2
↦ //        block   1 : ordering lp
↦ //                  : names    x y
↦ //        block   2 : ordering C
poly f = x5+y11+xy9+x3y9;
ideal i = jacob(f);
ideal j = i,i[1]*i[2]+t*x5y8;  // deformed ideal, parameter t
vdim(std(j));
↦ 40
ring R=0,(x,y),lp;
ideal i=imap(Rt,i);
int a=random(1,30000);
ideal j=i,i[1]*i[2]+a*x5y8;  // deformed ideal, fixed integer a
vdim(std(j));
↦ 40
```

A.7 T1 and T2

T1 resp. T2 of an ideal j do usually denote the vector spaces of infinitesimally deformations resp. of obstructions. In SINGULAR there are procedures `T1` and `T2` in `sing.lib` to compute this. `T1(j)` and `T2(j)` compute a standard basis of a presentation of these modules. A basis of the vectorspaces T1 resp. T2 is computed by applying `kbase`: `kbase(T1(j));` resp. `kbase(T2(j));`, the dimensions by applying `vdim`. For a complete intersection j the procedure `Tjurina` does also compute T1, but faster (T2=0 in this case). For a non complete intersection, it is faster to use the procedure `T12` instead of `T1` and `T2`. Type `help T1;` (or `help T2;` or `help T12;`) to obtain more detailed information about these procedures.

We give three examples, the first being a hypersurface, the second a complete intersection, the third no complete intersection:

- load `sing.lib`
- check wether the ideal j is a complete intersection. It is, if number of variables = dimension + minimal number of generators
- compute the Tjurina number
- compute a vectorspace basis (kbase) of T1
- compute the Hilbert function of T1
- create a polynomial encoding the Hilbert series
- compute the dimension of T2

```
LIB "sing.lib";
ring R=32003,(x,y,z),ds;

// ----------------------------------------
// hypersurface case (from series T[p,q,r]):
int p,q,r = 3,3,4;
poly f = x^p+y^q+z^r+xyz;
tjurina(f);
↦ 8
kbase(Tjurina(f));
↦ // Tjurina number = 8
↦ _[1]=z3
↦ _[2]=z2
↦ _[3]=yz
↦ _[4]=xz
↦ _[5]=z
↦ _[6]=y
↦ _[7]=x
↦ _[8]=1
// Tjurina number = 8

// ----------------------------------------
// complete intersection case (from series P[k,l]):
int k,l =3,2;
ideal j=xy,x^k+y^l+z2;
dim(std(j));          // Krull dimension
↦ 1
size(minbase(j));     // minimal number of generators
↦ 2
tjurina(j);           // Tjurina number
↦ 6
module T=Tjurina(j);
↦ // Tjurina number = 6
kbase(T);             // a sparse output of the k-basis of T1
↦ _[1]=z*gen(1)
↦ _[2]=gen(1)
↦ _[3]=y*gen(2)
↦ _[4]=x2*gen(2)
```

```
↦ _[5]=x*gen(2)
↦ _[6]=gen(2)
print(kbase(T));      // columns of matrix are a k-basis of T1
↦ z,1,0,0, 0,0,
↦ 0,0,y,x2,x,1

// ----------------------------------------
// general case (cone over rational normal curve of degree 4):
ring r1=0,(x,y,z,u,v),ds;
matrix m[2][4]=x,y,z,u,y,z,u,v;
ideal i=minor(m,2);   // 2x2 minors of matrix m
module M=T1(i);       // a presentation matrix of T1
↦ // dim T1 = 4
vdim(M);              // Tjurina number
↦ 4
hilb(M);              // display of both Hilbert series
↦ //         4 t^0
↦ //       -20 t^1
↦ //        40 t^2
↦ //       -40 t^3
↦ //        20 t^4
↦ //        -4 t^5
↦
↦ //         4 t^0
↦ // codimension  = 5
↦ // dimension    = 0
↦ // multiplicity = 4
intvec v1=hilb(M,1);  // first Hilbert series as intvec
intvec v2=hilb(M,2);  // second Hilbert series as intvec
v1;
↦ 4,-20,40,-40,20,-4,0
v2;
↦ 4,0
v1[3];                // 3-rd coefficient of the 1-st Hilbert series
↦ 40
module N=T2(i);
↦ // dim T2 = 3


// In some cases it might be useful to have a polynomial in some ring
// encoding the Hilbert series. This polynomial can then be
// differentiated, evaluated etc. It can be done as follows:
ring H = 0,t,ls;
poly h1;
int ii;
for (ii=1; ii<=size(v1); ii=ii+1)
{
   h1=h1+v1[ii]*t^(ii-1);
}
h1;                   // 1-st Hilbert series
↦ 4-20t+40t2-40t3+20t4-4t5
diff(h1,t);           // differentiate  h1
```

```
↦ -20+80t-120t2+80t3-20t4
subst(h1,t,1);        // substitute t by 1
↦ 0

// The procedures T1, T2, T12 may be called with two arguments and then
// they return a list with more information (type help T1; etc.)
// e.g. T12(i,<any>); returns a list with 9 nonempty objects where
// _[1] = std basis of T1-module, _[2] = std basis of T2-module,
// _[3]= vdim of T1, _[4]= vdim of T2
setring r1;           // make r1 again the basering
list L = T12(i,1);
↦ // dim T1  =  4
↦ // dim T2  =  3
kbase(L[1]);          // kbase of T1
↦ _[1]=1*gen(2)
↦ _[2]=1*gen(3)
↦ _[3]=1*gen(6)
↦ _[4]=1*gen(7)
kbase(L[2]);          // kbase of T2
↦ _[1]=1*gen(6)
↦ _[2]=1*gen(8)
↦ _[3]=1*gen(9)
L[3];                 // vdim of T1
↦ 4
L[4];                 // vdim of T2
↦ 3
```

A.8 Deformations

- The libraries `sing.lib` resp. `deform.lib` contain procedures to compute the miniversal (= semiuniversal deformation) of an isolated complete intersection singularity resp. arbitrary isolated singularity.
- The procedure `deform` in `sing.lib` returns a matrix whose columns represent a basis of 1-st order miniversal deformations.
- The procedure `versal` in `deform.lib` computes a formal miniversal deformation up to a certain order which can be prescribed by the user. For a complete intersection the 1-st order part is already miniversal.
- The procedure `versal` extends the basering to a new ring with additional deformation parameters which contains the equations for the miniversal base space and the miniversal total space.
- There are default names for the objects created, but the user may also choose own names.
- If the user sets `printlevel=2;` before running `versal`, some intermediate results are shown. This is useful since `versal` is already complicated and since it might run quite long on more complicated examples. (type `help versal;`)

We compute for the same examples as in the preceeding section the miniversal deformations:

```
LIB "deform.lib";
ring R=32003,(x,y,z),ds;
```

```
//------------------------------------------------
// hypersurface case (from series T[p,q,r]):
int p,q,r = 3,3,4;
poly f = x^p+y^q+z^r+xyz;
print(deform(f));
↦ z3,z2,yz,xz,z,y,x,1
// the miniversal deformation of f=0 is the projection from the
// miniversal total space to the miniversal base space:
// { (A,B,C,D,E,F,G,H,x,y,z) | x3+y3+xyz+z4+A+Bx+Cxz+Dy+Eyz+Fz+Gz2+Hz3 =0 }
//  --> { (A,B,C,D,E,F,G,H) }
//------------------------------------------------
// complete intersection case (from series P[k,l]):
int k,l =3,2;
ideal j=xy,x^k+y^l+z2;
print(deform(j));
↦ 0,0, 0,0,z,1,
↦ y,x2,x,1,0,0
versal(j);                    // using default names
↦ // smooth base space
↦ // ready: T1 and T2
↦
↦ // Result belongs to ring Px.
↦ // Equations of total space of miniversal deformation are
↦ // given by Fs, equations of miniversal base space by Js.
↦ // Make Px the basering and list objects defined in Px by typing:
↦    setring Px; show(Px);
↦    listvar(matrix);
↦ // NOTE: rings Qx, Px, So are alive!
↦ // (use 'kill_rings("");' to remove)
setring Px;
show(Px);                     // show is a procedure from inout.lib
↦ // ring: (32003),(A,B,C,D,E,F,x,y,z),(ds(6),ds(3),C);
↦ // minpoly = 0
listvar(ideal);
// ___ Equations of miniversal base space ___:
Js;
↦
// ___ Equations of miniversal total space ___:
Fs;
↦ Fs[1,1]=xy+Ez+F
↦ Fs[1,2]=y2+z2+x3+Ay+Bx2+Cx+D
// the miniversal deformation of V(j) is the projection from the
// miniversal total space to the miniversal base space:
// { (A,B,C,D,E,F,x,y,z) | xy+A+Bz=0, y2+z2+x3+C+Dx+Ex2+Fy=0 }
//  --> { (A,B,C,D,E,F) }
//------------------------------------------------
// general case (cone over rational normal curve of degree 4):
ring r1=0,(x,y,z,u,v),ds;
matrix m[2][4]=x,y,z,u,y,z,u,v;
ideal i=minor(m,2);                 // 2x2 minors of matrix m
int time=timer;
```

```
// Def_r is the name of the miniversal base space with
// parameters A(1),...,A(4)
versal(i,0,"Def_r","A(");
↦ // ready: T1 and T2
↦
↦ // Result belongs to ring Def_rPx.
↦ // Equations of total space of miniversal deformation are
↦ // given by Fs, equations of miniversal base space by Js.
↦ // Make Def_rPx the basering and list objects defined in Def_rPx by t
↦ yping:
↦    setring Def_rPx; show(Def_rPx);
↦    listvar(matrix);
↦ // NOTE: rings Def_rQx, Def_rPx, Def_rSo are alive!
↦ // (use 'kill_rings("Def_r");' to remove)
"// used time:",timer-time,"sec";   // time for miniversal
↦ // used time: 0 sec
// the miniversal deformation of V(i) is the projection from the
// miniversal total space to the miniversal base space:
// { (A(1..4),x,y,z,u,v) |
//         -y^2+x*z+A(2)*x-A(3)*y=0, -y*z+x*u-A(1)*x-A(3)*z=0,
//         -y*u+x*v-A(3)*u-A(4)*z=0, -z^2+y*u-A(1)*y-A(2)*z=0,
//         -z*u+y*v-A(2)*u-A(4)*u=0, -u^2+z*v+A(1)*u-A(4)*v=0 }
//  --> { A(1..4) |
//         -A(1)*A(4) = A(3)*A(4) = -A(2)*A(4)-A(4)^2 = 0 }
//------------------------------------------------------
```

A.9 Finite fields

We define a variety in n-space of codimension 2 defined by polynomials of degree d with generic coefficients over the prime field Z/p and look for zeroes on the torus. First over the prime field and then in the finite extension field with p^k elements. In general there will be many more solutions in the second case. (Since the SINGULAR language is interpreted, the evaluation of many `for`-loops is not very fast):

```
int p=3;  int n=3;  int d=5; int k=2;
ring rp = p,(x(1..n)),dp;
int s = size(maxideal(d));
s;
↦ 21
// create a dense homogeneous ideal m, all generators of degree d, with
// generic (random) coefficients:
ideal m = maxideal(d)*random(p,s,n-2);
m;
↦ m[1]=x(1)^3*x(2)^2-x(1)*x(2)^4+x(1)^4*x(3)-x(1)^3*x(2)*x(3)+x(1)*x(2)
↦ ^3*x(3)+x(2)^4*x(3)+x(2)^3*x(3)^2+x(1)*x(2)*x(3)^3+x(1)*x(3)^4-x(3)^5
↦
// look for zeroes on the torus by checking all points (with no component 0)
// of the affine n-space over the field with p elements :
ideal mt;
int i(1..n);                    // initialize integers i(1),...,i(n)
```

```
int l;
s=0;
for (i(1)=1;i(1)<p;i(1)=i(1)+1)
{
  for (i(2)=1;i(2)<p;i(2)=i(2)+1)
  {
    for (i(3)=1;i(3)<p;i(3)=i(3)+1)
    {
      mt=m;
      for (l=1;l<=n;l=l+1)
      {
        mt=subst(mt,x(l),i(l));
      }
      if (size(mt)==0)
      {
        "solution:",i(1..n);
        s=s+1;
      }
    }
  }
}
↦ solution: 1 1 2
↦ solution: 1 2 1
↦ solution: 1 2 2
↦ solution: 2 1 1
↦ solution: 2 1 2
↦ solution: 2 2 1
"//",s,"solutions over GF("+string(p)+")";
↦ // 6 solutions over GF(3)
// Now define a ring rpk with p^k elements, call the primitive element z.
// Hence 'solution exponent: 0 1 5' means that (z^0,z^1,z^5) is a
// solution
ring rpk=(p^k,z),(x(1..n)),dp;
rpk;
↦ //   # ground field : 9
↦ //   primitive element : z
↦ //   minpoly        : 1*z^2+1*z^1+2*z^0
↦ //   number of vars : 3
↦ //        block   1 : ordering dp
↦ //                  : names    x(1) x(2) x(3)
↦ //        block   2 : ordering C
ideal m=imap(rp,m);
s=0;
ideal mt;
for (i(1)=0;i(1)<p^k-k;i(1)=i(1)+1)
{
  for (i(2)=0;i(2)<p^k-k;i(2)=i(2)+1)
  {
    for (i(3)=0;i(3)<p^k-k;i(3)=i(3)+1)
    {
      mt=m;
```

```
      for (l=1;l<=n;l=l+1)
      {
        mt=subst(mt,x(l),z^i(l));
      }
      if (size(mt)==0)
      {
        "solution exponent:",i(1..n);
        s=s+1;
      }
    }
  }
}
↦ solution exponent: 0 0 2
↦ solution exponent: 0 0 4
↦ solution exponent: 0 0 6
↦ solution exponent: 0 1 0
↦ solution exponent: 0 3 0
↦ solution exponent: 0 4 0
↦ solution exponent: 0 4 4
↦ solution exponent: 0 4 5
↦ solution exponent: 1 1 3
↦ solution exponent: 1 1 5
↦ solution exponent: 1 2 1
↦ solution exponent: 1 4 1
↦ solution exponent: 1 5 0
↦ solution exponent: 1 5 1
↦ solution exponent: 1 5 5
↦ solution exponent: 1 5 6
↦ solution exponent: 2 2 0
↦ solution exponent: 2 2 4
↦ solution exponent: 2 2 6
↦ solution exponent: 2 3 2
↦ solution exponent: 2 5 2
↦ solution exponent: 2 6 1
↦ solution exponent: 2 6 2
↦ solution exponent: 2 6 6
↦ solution exponent: 3 3 1
↦ solution exponent: 3 3 5
↦ solution exponent: 3 4 3
↦ solution exponent: 3 6 3
↦ solution exponent: 4 0 0
↦ solution exponent: 4 0 1
↦ solution exponent: 4 0 3
↦ solution exponent: 4 0 4
↦ solution exponent: 4 4 0
↦ solution exponent: 4 4 2
↦ solution exponent: 4 4 6
↦ solution exponent: 4 5 4
↦ solution exponent: 5 0 5
↦ solution exponent: 5 1 1
↦ solution exponent: 5 1 2
```

```
↦ solution exponent: 5 1 4
↦ solution exponent: 5 1 5
↦ solution exponent: 5 5 1
↦ solution exponent: 5 5 3
↦ solution exponent: 5 6 5
↦ solution exponent: 6 1 6
↦ solution exponent: 6 2 2
↦ solution exponent: 6 2 3
↦ solution exponent: 6 2 5
↦ solution exponent: 6 2 6
↦ solution exponent: 6 6 0
↦ solution exponent: 6 6 2
↦ solution exponent: 6 6 4
"//",s,"solutions over GF("+string(p^k)+")";
↦ // 52 solutions over GF(9)
```

A.10 Elimination

Elimination is the algebraic counterpart of the geometric concept of projection. If $f = (f_1, \ldots, f_n) : k^r \to k^n$ is a polynomial map, the Zariski-closure of the image is the zero-set of the ideal

$$j = J \cap k[x_1, \ldots, x_n], \quad \text{where}$$
$$J = (x_1 - f_1(t_1, \ldots, t_r), \ldots, x_n - f_n(t_1, \ldots, t_r)) \subseteq k[t_1, \ldots, t_r, x_1, \ldots, x_n]$$

i.e of the ideal j obtained from J by eliminating the variables $t_1, \ldots, t_r$. This can be done by computing a standard basis of J with respect to a product ordering where the block of t-variables comes before the block of x-variables and then selecting those polynomials which do not contain any t. In SINGULAR the most convenient way is to use the `eliminate` command. In contrast to the first method, with `eliminate` the result need not be a standard basis in the given ordering. Hence, there may be cases where the first method is to be preferred.

WARNING: In the case of a local or a mixed ordering, elimination needs special care. f may be considered as a map of germs $f : (k^r, 0) \to (k^n, 0)$, but even if this map germ is finite, we are in general not able to compute the image germ because for this we would need an implementation of the Weierstrass preparation theorem. What we can compute, and what `eliminate` actually does, is the following. Let V(J) be the zero-set of J in $k^r \times (k^n, 0)$, then the closure of the image of V(J) under the projection

$$\text{pr} : k^r \times (k^n, 0) \to (k^n, 0)$$

can be computed. Note that this germ contains also those components of V(J) which meet the fibre of pr but not necessarily in the origin. This is achieved by an ordering with the block of t-variables having a global ordering (and coming before the x-variables) and the x-variables having a local ordering. In a local situation we propose `eliminate` with ordering ls.

In any case, if the input is weighted homogeneous (=quasihomogeneous), the variables should be given the correct weights. SINGULAR offers a function `weight` which proposes, given an ideal or module, integer weights for the variables, such that the ideal resp. module is as homogeneous as possible with respect to these weights. The function finds correct weights, if the input is weighted homogeneous (but is rather slow for many variables). In order to check, whether the input is quasihomogeneous, use the function `qhweight`, which returns an intvec of correct weights if the input is quasihomogeneous and an intvec of zeroes otherwise.

Let us give two examples:

1. First we compute the equations of the simple space curve T[7]′ consisting of two tangential cusps given in parametric form.
2. We compute weights for the equations and check that the equations are quasihomogeneous with these weights.
3. Then we compute the tangent developable of the rational normal curve in P^4.

```
// 1. Compute equations of curve given in parametric form:
// Two transversal cusps in (k^3,0):
ring r1 = 0,(t,x,y,z),ls;
ideal i1 = x-t2,y-t3,z;         // parametrization of the first branch
ideal i2 = y-t2,z-t3,x;         // parametrization of the second branch
ideal j1 = eliminate(i1,t);
j1;                             // equations of the first branch
↦ j1[1]=z
↦ j1[2]=y2-x3
ideal j2 = eliminate(i2,t);
j2;                             // equations of the second branch
↦ j2[1]=x
↦ j2[2]=z2-y3
// Now map to a ring with only x,y,z as variables and compute the
// intersection of j1 and j2 there:
ring r2 = 0,(x,y,z),ds;
ideal j1= imap(r1,j1);          // imap is a convenient ringmap for
ideal j2= imap(r1,j2);          // inclusions and projections of rings
ideal i = intersect(j1,j2);
i;                              // equations of both branches
↦ i[1]=z2-y3+x3y
↦ i[2]=xz
↦ i[3]=xy2-x4
↦ i[4]=x3z
//
// 2. Compute the weights:
intvec v= qhweight(i);          // compute weights
v;
↦ 4,6,9
//
// 3. Compute the tangent developable
// The tangent developable of a projective variety given parametrically
// by F=(f1,...,fn) : P^r --> P^n is the union of all tangent spaces
// of the image. The tangent space at a smooth point F(t1,...,tr)
// is given as the image of the tangent space at (t1,...,tr) under
// the tangent map (affine coordinates)
//   T(t1,...,tr): (y1,...,yr) --> jacob(f)*transpose((y1,...,yr))
// where jacob(f) denotes the jacobian matrix of f with respect to the
// t's evaluated at the point (t1,...,tr).
// Hence we have to create the graph of this map and then to eliminate
// the t's and y's.
// The rational normal curve in P^4 is given as the image of
//        F(s,t) = (s4,s3t,s2t2,st3,t4)
```

```
// each component being homogeneous of degree 4.
ring P = 0,(s,t,x,y,a,b,c,d,e),dp;
ideal M = maxideal(1);
ideal F = M[1..2];     // take the 1-st two generators of M
F=F^4;
// simplify(...,2); deletes 0-columns
matrix jac = simplify(jacob(F),2);
ideal T = x,y;
ideal J = jac*transpose(T);
ideal H = M[5..9];
ideal i = H-J;         // this is tricky: difference between two
                       // ideals is not defined, but between two
                       // matrices. By automatic type conversion
                       // the ideals are converted to matrices,
                       // subtracted and afterwards converted
                       // to an ideal. Note that '+' is defined
                       // and adds (concatenates) two ideals
i;
↦ i[1]=-4s3x+a
↦ i[2]=-3s2tx-s3y+b
↦ i[3]=-2st2x-2s2ty+c
↦ i[4]=-t3x-3st2y+d
↦ i[5]=-4t3y+e
// Now we define a ring with product ordering and weights 4
// for the variables a,...,e.
// Then we map i from P to P1 and eliminate s,t,x,y from i.
ring P1 = 0,(s,t,x,y,a,b,c,d,e),(dp(4),wp(4,4,4,4,4));
ideal i = fetch(P,i);
ideal j= eliminate(i,stxy);    // equations of tangent developable
j;
↦ j[1]=3c2-4bd+ae
↦ j[2]=2bcd-3ad2-3b2e+4ace
↦ j[3]=8b2d2-9acd2-9b2ce+14abde-4a2e2
// We can use the product ordering to eliminate s,t,x,y from i
// by a std-basis computation.
// We need proc 'nselect' from elim.lib.
LIB "elim.lib";
j = std(i);                    // compute a std basis j
j = nselect(j,1,4);            // select generators from j not
j;                             // containing variable 1,...,4
↦ j[1]=3c2-4bd+ae
↦ j[2]=2bcd-3ad2-3b2e+4ace
↦ j[3]=8b2d2-9acd2-9b2ce+12ac2e-2abde
```

A.11 Free resolution

In SINGULAR a free resolution of a module or ideal has its own type: resolution. It is a structure that stores all information related to free resolutions. This allows partial computations of resolutions via the command `res`. After applying `res`, only a pre-format of the resolution is computed which allows to determine invariants like Betti-numbers or homological dimension. To see the differentials

of the complex, a resolution must be converted into the type list which yields a list of modules: the k-th module in this list is the first syzygy-module (module of relations) of the (k-1)st module. There are the following commands to compute a resolution:

`res` Section 5.1.96 [res], page 162
computes a free resolution of an ideal or module using a heuristically chosen method. This is the prefered method to compute free resolutions of ideals or modules.

`lres` Section 5.1.59 [lres], page 139
computes a free resolution of an ideal or module with LaScala's method. The input needs to be homogeneous.

`mres` Section 5.1.67 [mres], page 144
computes a minimal free resolution of an ideal or module with the Szyzygy method.

`sres` Section 5.1.104 [sres], page 167
computes a free resolution of an ideal or module with Schreyer's method. The input has to be a standard basis.

`nres` Section 5.1.74 [nres], page 148
computes a free resolution of an ideal or module with the standard basis method.

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

`syz` Section 5.1.111 [syz], page 173
computes the first Szyzygy (i.e., the module of relations of the given generators).

`res(i,r)`, `lres(i,r)`, `sres(i,r)`, `mres(i,r)`, `nres(i,r)` compute the first r modules of the resolution of i, resp. the full resolution if r=0 and the basering is not a qring. See the the manual for a precise description of these commands.
Note: The command `betti` does not require a minimal resolution for the minimal betti numbers.

Now let's look at an examples which uses resolutions: The Hilbert-Burch theorem says that the ideal i of a reduced curve in K^3 has a free resolution of length 2 and that i is given by the 2x2 minors of the 2nd matrix in the resolution. We test this for the two transversal cusps in K^3. Afterwards we compute the resolution of the ideal j of the tangent developable of the rational normal curve in P^4 from above. Finally we demonstrate the use of the type resoltuion in connection with the lres command.

```
// Two transversal cusps in (k^3,0):
ring r2 =0,(x,y,z),ds;
ideal i =z2-1y3+x3y,xz,-1xy2+x4,x3z;
resolution rs=mres(i,0);   // computes a minimal resolution
rs;                        // the standard representation of complexes
↦   1       3       2       0
↦ r2  <-- r2  <-- r2  <-- r2
↦
↦ 0       1       2       3
↦
  list resi=rs;            // convertion to a list
print(resi[1]);            // the 1-st module is i minimized
↦ xz,
↦ z2-y3+x3y,
↦ xy2-x4
print(resi[2]);            // the 1-st syzygy module of i
```

```
↦ -z,-y2+x3,
↦ x, 0,
↦ y, z
resi[3];                    // the 2-nd syzygy module of i
↦ _[1]=0
ideal j=minor(resi[2],2);
reduce(j,std(i));           // check whether j is contained in i
↦ _[1]=0
↦ _[2]=0
↦ _[3]=0
size(reduce(i,std(j)));     // check whether i is contained in j
↦ 0
// size(<ideal>) counts the non-zero generators
// ------------------------------------------
// The tangent developable of the rational normal curve in P^4:
ring P = 0,(a,b,c,d,e),dp;
ideal j= 3c2-4bd+ae, -2bcd+3ad2+3b2e-4ace,
         8b2d2-9acd2-9b2ce+9ac2e+2abde-1a2e2;
resolution rs=mres(j,0);
rs;
↦  1      2      1      0
↦ P  <-- P  <-- P  <-- P
↦
↦ 0      1      2      3
↦
list L=rs;
print(L[2]);
↦ 2bcd-3ad2-3b2e+4ace,
↦ -3c2+4bd-ae
// create an intmat with graded betti numbers
intmat B=betti(rs);
// this gives a nice output of betti numbers
print(B,"betti");
↦            0     1     2
↦ ------------------------
↦     0:     1     0     0
↦     1:     0     1     0
↦     2:     0     1     0
↦     3:     0     0     1
↦ ------------------------
↦ total:     1     2     1
// the user has access to all betti numbers
// the 2-nd column of B:
B[1..4,2];
↦ 0 1 1 0
ring cyc5=32003,(a,b,c,d,e,h),dp;
ideal i=
a+b+c+d+e,
ab+bc+cd+de+ea,
abc+bcd+cde+dea+eab,
abcd+bcde+cdea+deab+eabc,
```

```
h5-abcde;
resolution rs=lres(i,0);   //computes the resolution according LaScala
rs;                        //the shape of the minimal resolution
↦     1         5         10         10         5         1         0
↦ cyc5  <-- cyc5  <-- cyc5   <-- cyc5   <-- cyc5  <-- cyc5  <-- cyc5
↦
↦ 0         1         2          3          4         5         6
↦ resolution not minimized yet
↦
print(betti(rs),"betti");  //shows the Betti-numbers of cyclic 5
↦            0     1     2     3     4     5
↦ ------------------------------------------
↦     0:     1     1     0     0     0     0
↦     1:     0     1     1     0     0     0
↦     2:     0     1     1     0     0     0
↦     3:     0     1     2     1     0     0
↦     4:     0     1     2     1     0     0
↦     5:     0     0     2     2     0     0
↦     6:     0     0     1     2     1     0
↦     7:     0     0     1     2     1     0
↦     8:     0     0     0     1     1     0
↦     9:     0     0     0     1     1     0
↦    10:     0     0     0     0     1     1
↦ ------------------------------------------
↦ total:     1     5    10    10     5     1
dim(rs);                   //the homological dimension
↦ 4
size(list(rs));            //gets the full (non-reduced) resolution
↦ 6
minres(rs);                //minimizes the resolution
↦     1         5         10         10         5         1         0
↦ cyc5  <-- cyc5  <-- cyc5   <-- cyc5   <-- cyc5  <-- cyc5  <-- cyc5
↦
↦ 0         1         2          3          4         5         6
↦
size(list(rs));            //gets the minimized resolution
↦ 6
```

A.12 Ext

We start by showing how to calculate the n-th Ext group of an ideal. The ingredients to do this, are by the definition of Ext the following: calculate a (minimal) resolution at least up to length n, apply the Hom-functor, and calculate the n-th Homology group, so form the quotient ker/Im in the resolution sequence.

The Hom functor is given simply by transposing (hence dualizing) the module or the corresponding matrix with the command `transpose`. The image of the $n-1$-st map is generated by the columns of the corresponding matrix. To calculate the kernel apply the command `syz` at the $n-1$-st transposed entry of the resolution. Finally, the quotient is obtained by the command

`modulo`, which gives for two modules A = ker, B = Im the module of relations of $A/(A \cap B)$ in the usual way. As we have a chain complex this is obviously the same as ker/Im.

We collect these statements in this short procedure.

```
proc ext(int n, ideal I)
{
  resolution rs = mres(I,n+1);
  module tAn    = transpose(rs[n+1]);
  module tAn_1  = transpose(rs[n]);
  module ext_n  = modulo(syz(tAn),tAn_1);
  return(ext_n);
}
```

Now consider the following example:

```
ring r5 = 32003,(a,b,c,d,e),dp;
ideal I = a2b2+ab2c+b2cd, a2c2+ac2d+c2de,a2d2+ad2e+bd2e,a2e2+abe2+bce2;
print(ext(2,I));
↦ 1,0,0,0,0,0,0,
↦ 0,1,0,0,0,0,0,
↦ 0,0,1,0,0,0,0,
↦ 0,0,0,1,0,0,0,
↦ 0,0,0,0,1,0,0,
↦ 0,0,0,0,0,1,0,
↦ 0,0,0,0,0,0,1
ext(3,I);   // too big to be displayed here
```

The library `homolog.lib` contains several procedures for computing Ext-modules and related modules, which are much more general and sophsticated then the above one. They are used in the following example.

If M is a module, then $\mathrm{Ext}^1(M,M)$ resp. $\mathrm{Ext}^2(M,M)$ are the modules of infinitesimal deformations resp. of obstructions of M (like T1 and T2 for a singularity). Similar to the treatment for singularities, the semiuniversal deformation of M can be computed (if Ext^1 is finite dimensional) with the help of Ext^1, Ext^2 and the cup product. There is an extra procedure for $\mathrm{Ext}^k(R/J,R)$ if J is an ideal in R since this is faster than the general Ext.

We compute

- the infinitesimal deformations ($= \mathrm{Ext}^1(K,K)$) and obstructions ($= \mathrm{Ext}^2(K,K)$) of the residue field $K = R/m$ of an ordinary cusp, R=$\mathrm{Loc}_K[x,y]/(x^2-y^3)$, m=$(x,y)$. For $\mathrm{Ext}^1(m,m)$ here we have to compute `Ext(1,syz(m),syz(m))` with `syz(m)` the first syzygy module of m, which is isomorphic to $\mathrm{Ext}^2(K,K)$.
- $\mathrm{Ext}^k(R/i,R)$ for some ideal i and with an extra option

```
LIB "homolog.lib";
ring R=0,(x,y),ds;
ideal i=x2-y3;
qring q = std(i);          // defines the quotient ring Loc_k[x,y]/(x2-y3)
```

```
ideal m = maxideal(1);
module T1K = Ext(1,m,m);   // computes Extˆ1(R/m,R/m)
↦ // degree of Extˆ1:
↦ // codimension  = 2
↦ // dimension    = 0
↦ // multiplicity = 2
↦
print(T1K);
↦ 0,  0,y,x,0,y,0,    x2-y3,
↦ -y2,x,x,0,y,0,x2-y3,0,
↦ 1,  0,0,0,0,0,0,    0
printlevel=2;               // gives more explanation
module T2K=Ext(2,m,m);      // computes Extˆ2(R/m,R/m)
↦ // Computing Extˆ2 (help Ext; gives an explanation):
↦ // Let 0<--coker(M)<--F0<--F1<--F2<--... be a resolution of coker(M),
↦
↦ // and 0<--coker(N)<--G0<--G1 a presentation of coker(N),
↦ // then Hom(F2,G0)-->Hom(F3,G0) is given by:
↦ y2,x,
↦ x, y
↦ // and Hom(F1,G0) + Hom(F2,G1)-->Hom(F2,G0) is given by:
↦ -y,x,  x,0,y,0,
↦ x, -y2,0,x,0,y
↦
↦ // degree of Extˆ2:
↦ // codimension  = 2
↦ // dimension    = 0
↦ // multiplicity = 2
↦
print(std(T2K));
↦ -y2,0,x, 0,y,
↦ 0,  x,-y,y,0,
↦ 1,  0,0, 0,0
printlevel=0;
module E = Ext(1,syz(m),syz(m));
↦ // degree of Extˆ1:
↦ // codimension  = 2
↦ // dimension    = 0
↦ // multiplicity = 2
↦
print(std(E));
↦ -y,x, 0, 0,0,x, 0,y,
↦ 0, -y,-y,0,x,-y,y,0,
↦ 0, 0, 0, 1,0,0, 0,0,
↦ 0, 0, 1, 0,0,0, 0,0,
↦ 0, 1, 0, 0,0,0, 0,0,
↦ 1, 0, 0, 0,0,0, 0,0
//We see from the matrices that T2K and E are isomorphic
//as it should be; but both are differently presented
//----------------------------------------
```

```
 ring S=0,(x,y,z),dp;
ideal  i = x2y,y2z,z3x;
module E = Ext_R(2,i);
↦ // degree of Ext^2:
↦ // codimension = 2
↦ // dimension   = 1
↦ // degree      = 7
↦
print(E);
↦ 0,y,0,z2,
↦ z,0,0,-x,
↦ 0,0,x,-y
// if a 3-rd argument is given (of any type)
// a list of Ext^k(R/i,R), a SB of Ext^k(R/i,R) and a vector space basis
// is returned:
list LE = Ext_R(3,i,"");
↦ // degree of Ext^3:
↦ // codimension = 3
↦ // dimension   = 0
↦ // degree      = 2
↦
LE;
↦ [1]:
↦    _[1]=y*gen(1)
↦    _[2]=x*gen(1)
↦    _[3]=z2*gen(1)
↦ [2]:
↦    _[1]=y*gen(1)
↦    _[2]=x*gen(1)
↦    _[3]=z2*gen(1)
↦ [3]:
↦    _[1,1]=z
↦    _[1,2]=1
print(LE[2]);
↦ y,x,z2
print(kbase(LE[2]));
↦ z,1
```

A.13 Polar curves

The polar curve of a hypersurface given by a polynomial $f \in k[x_1, \ldots, x_n, t]$ with respect to t (we may consider $f = 0$ as a family of hypersurfaces parametrized by t) is defined as the Zariski closure of $V(\partial f/\partial x_1, \ldots, \partial f/\partial x_n) \setminus V(f)$ if this happens to be a curve. Some authors consider $V(\partial f/\partial x_1, \ldots, \partial f/\partial x_n)$ itself as polar curve.

We may consider projective hypersurfaces (in P^n), affine hypersurfaces (in k^n) or germs of hypersurfaces (in $(k^n, 0)$), getting in this way projective, affine or local polar curves.

We shall compute this now for a family of plane curves. We need the library `elim.lib` for saturation and `sing.lib` for the singular locus.

```
LIB "elim.lib";
LIB "sing.lib";
// Affine polar curve:
ring R = 0,(x,z,t),dp;              // global ordering dp
poly f = z5+xz3+x2-tz6;
dim_slocus(f);                      // dimension of singular locus
↦ 1
ideal j = diff(f,x),diff(f,z);
dim(std(j));                        // dim V(j)
↦ 1
dim(std(j+ideal(f)));               // V(j,f) also 1-dimensional
↦ 1
// j defines a curve, but to get the polar curve we must remove the
// branches contained in f=0 (they exist since dim V(j,f) = 1). This
// gives the polar curve set theoretically. But for the structure we
// may take either j:f or j:f^k for k sufficiently large. The first is
// just the ideal quotient, the second the iterated ideal quotient
// or saturation. In our case both is the same.
ideal q = quotient(j,ideal(f));     // ideal quotient
ideal qsat = sat(j,f)[1];           // saturation, proc from elim.lib
ideal sq = std(q);
dim(sq);
↦ 1
// 1-dimensional, hence q defines the affine polar curve
//
// to check that q and qsat are the same, we show both inclusions, i.e.
// both reductions must give the 0-ideal
size(reduce(qsat,sq));
↦ 0
size(reduce(q,std(qsat)));
↦ 0
qsat;
↦ qsat[1]=12zt+3z-10
↦ qsat[2]=5z2+12xt+3x
↦ qsat[3]=144xt2+72xt+9x+50z
// We see that the affine polar curve does not pass through the origin,
// hence we expect the local polar "curve" to be empty
// ------------------------------------------------

// Local polar curve:
ring r = 0,(x,z,t),ds;              // local ordering ds
poly f = z5+xz3+x2-tz6;
dim_slocus(f);                      // dimension of singular locus
↦ 1
ideal j = diff(f,x),diff(f,z);
dim(std(j));                        // V(j) 1-dimensional
↦ 1
dim(std(j+ideal(f)));               // V(j,f) also 1-dimensional
↦ 1
ideal q = quotient(j,ideal(f));     // ideal quotient
q;
```

```
↦ q[1]=1
// The local polar "curve" is empty, i.e V(j) is contained in V(f)
// ------------------------------------------------

// Projective polar curve: (we need "sing.lib" and "elim.lib")
ring P = 0,(x,z,t,y),dp;            // global ordering dp
poly f = z5+xz3+x2-tz6;
f = z5y+xz3y2+x2y4-tz6;             // homogenize f with respect to y
                                    // but consider t as parameter
dim_slocus(f);              // projective 1-dimensional singular locus
↦ 2
ideal j = diff(f,x),diff(f,z);
dim(std(j));                        // V(j), projective 1-dimensional
↦ 2
dim(std(j+ideal(f)));               // V(j,f) also projective 1-dimensional
↦ 2
ideal q = quotient(j,ideal(f));
ideal qsat = sat(j,f)[1];           // saturation, proc from elim.lib
dim(std(qsat));
↦ 2
// projective 1-dimensional, hence q and/or qsat define the projective
// polar curve. In this case, q and qsat are not the same, we needed
// 2 quotients.
// Let us check both reductions:
size(reduce(qsat,std(q)));
↦ 4
size(reduce(q,std(qsat)));
↦ 0
// Hence q is contained in qsat but not conversely
q;
↦ q[1]=12zty+3zy-10y2
↦ q[2]=60z2t-36xty-9xy-50zy
qsat;
↦ qsat[1]=12zt+3z-10y
↦ qsat[2]=12xty+5z2+3xy
↦ qsat[3]=144xt2+72xt+9x+50z
↦ qsat[4]=z3+2xy2
 // Now consider again the affine polar curve,
// homogenize it with respect to y (deg t=0) and compare:
// affine polar curve:
ideal qa = 12zt+3z-10,5z2+12xt+3x,-144xt2-72xt-9x-50z;
// homogenized:
ideal qh = 12zt+3z-10y,5z2+12xyt+3xy,-144xt2-72xt-9x-50z;
size(reduce(qh,std(qsat)));
↦ 0
size(reduce(qsat,std(qh)));
↦ 0
// both ideals coincide
```

A.14 Depth

We compute the depth of the module of Kaehler differentials $\mathrm{D}_k(\mathrm{R})$ of the variety defined by the $(m+1)$-minors of generic symmetric nxn matrix. We do this by computing the resolution over the polynomial ring. Then, by the Auslander-Buchsbaum formula, the depth is equal to the number of variable minus the length of a minimal resolution. This example was suggested by U. Vetter in order to check wheter his bound $\mathrm{depth}(\mathrm{D}_k(R)) \geq m(m+1)/2 + m - 1$ could be improved.

```
LIB "matrix.lib"; LIB "sing.lib";
int n = 4;
int m = 3;
int N = n*(n+1)/2;           // will become number of variables
ring R = 32003,x(1..N),dp;
matrix X = symmat(n);        // proc from matrix.lib
                             // creates the symmetric generic nxn matrix
print(X);
↦ x(1),x(2),x(3),x(4),
↦ x(2),x(5),x(6),x(7),
↦ x(3),x(6),x(8),x(9),
↦ x(4),x(7),x(9),x(10)
ideal J = minor(X,m);
J=std(J);
// Kaehler differentials D_k(R)
// of R=k[x1..xn]/J:
module D = J*freemodule(N)+transpose(jacob(J));
ncols(D);
↦ 110
nrows(D);
↦ 10
//
// Note: D is a submodule with 110 generators of a free module
// of rank 10 over a polynomial ring in 10 variables.
int time = timer;
module sD = std(D);
resolution Dres = sres(sD,0);               // the full resolution
timer-time;                               // time used for std + sres
↦ 1
intmat B = betti(Dres);
print(B,"betti");
↦            0     1     2     3     4     5     6
↦ ------------------------------------------------
↦     0:    10     0     0     0     0     0     0
↦     1:     0    10     0     0     0     0     0
↦     2:     0    84   144    60     0     0     0
↦     3:     0     0    35    80    60    16     1
↦ ------------------------------------------------
↦ total:    10    94   179   140    60    16     1
N-ncols(B)+1;                           // the desired depth
↦ 4
```

A.15 Formatting output

We show how to insert the result of a computation inside a text by using strings. First we compute the powers of 2 and comment the result with some text. Then we do the same and give the output a nice format by computing and adding appropriate space.

```
// The powers of 2:
int  n;
for (n = 2; n <= 128; n = n * 2)
{"n = " + string (n);}
↦ n = 2
↦ n = 4
↦ n = 8
↦ n = 16
↦ n = 32
↦ n = 64
↦ n = 128
// The powers of 2 in a nice format
int j;
string space = "";
for (n = 2; n <= 128; n = n * 2)
{
  space = "";
  for (j = 1; j <= 5 - size (string (n)); j = j+1)
  { space = space + " "; }
  "n =" + space + string (n);
}
↦ n =   2
↦ n =   4
↦ n =   8
↦ n =  16
↦ n =  32
↦ n =  64
↦ n = 128
```

A.16 Cyclic roots

We write a procedure returning a string that enables us to automatically create the ideal of cyclic roots over the basering with n variables. The procedure assumes that the variables consist of a single letter each (hence no indexed variables are allowed; the procedure `cyclic` in `poly.lib` does not have this restriction). Then we compute a standard basis of this ideal and some numerical information. (This ideal is used as a classical benchmark for standard basis computations).

```
// We call the procedure 'cyclic':
proc cyclic (int n)
{
   string vs = varstr(basering)+varstr(basering);
   int c=find(vs,",");
   while ( c!=0 )
   {
```

```
      vs=vs[1,c-1]+vs[c+1,size(vs)];
      c=find(vs,",");
   }
   string t,s;
   int i,j;
   for ( j=1; j<=n-1; j=j+1 )
   {
      t="";
      for ( i=1; i <=n; i=i+1 )
      {
         t = t + vs[i,j] + "+";
      }
      t = t[1,size(t)-1] + ","+newline;
      s=s+t;
   }
   s=s+vs[1,n]+"-1";
   return (s);
}

ring r=0,(a,b,c,d,e),lp;          // basering, char 0, lex ordering
string sc=cyclic(nvars(basering));
sc;                               // the string of the ideal
↦ a+b+c+d+e,
↦ ab+bc+cd+de+ea,
↦ abc+bcd+cde+dea+eab,
↦ abcd+bcde+cdea+deab+eabc,
↦ abcde-1
execute("ideal i="+sc+";");       // this defines the ideal of cyclic roots
i;
↦ i[1]=a+b+c+d+e
↦ i[2]=ab+bc+cd+ae+de
↦ i[3]=abc+bcd+abe+ade+cde
↦ i[4]=abcd+abce+abde+acde+bcde
↦ i[5]=abcde-1
timer=1;
ideal j=std(i);
↦ //used time: 7.5 sec
size(j);                          // number of elements in the std basis
↦ 11
degree(j);
↦ // codimension = 5
↦ // dimension   = 0
↦ // degree      = 70
```

A.17 G_a -Invariants

We work in characteristic 0 and use the Lie algebra generated by one vectorfield of the form $\sum x_i \partial/\partial x_{i+1}$.

```
LIB "invar.lib";
```

```
int n=5;
int i;
ring s=32003,(x(1..n)),wp(1,2,3,4,5);
// definition of the vectorfield m=sum m[i,1]*d/dx(i)
matrix m[n][1];
for (i=1;i<=n-1;i=i+1)
{
   m[i+1,1]=x(i);
}
// computation of the ring of invariants
ideal in=invariantRing(m,x(2),x(1));
in;   //invariant ring is generated by 5 invariants
↦ in[1]=x(1)
↦ in[2]=x(2)^2-2*x(1)*x(3)
↦ in[3]=x(3)^2-2*x(2)*x(4)+2*x(1)*x(5)
↦ in[4]=x(2)^3-3*x(1)*x(2)*x(3)+3*x(1)^2*x(4)
↦ in[5]=x(3)^3-3*x(2)*x(3)*x(4)-15997*x(1)*x(4)^2+3*x(2)^2*x(5)-6*x(1)*
↦ x(3)*x(5)
ring q=32003,(x,y,z,u,v,w),dp;
matrix m[6][1];
m[2,1]=x;
m[3,1]=y;
m[5,1]=u;
m[6,1]=v;
// the vectorfield is: xd/dy+yd/dz+ud/dv+vd/dw
ideal in=invariantRing(m,y,x);
in; //invariant ring is generated by 6 invariants
↦ in[1]=x
↦ in[2]=u
↦ in[3]=v2-2uw
↦ in[4]=zu-yv+xw
↦ in[5]=yu-xv
↦ in[6]=y2-2xz
```

A.18 Invariants of a finite group

Two algorithms to compute the invariant ring are implemented in SINGULAR, `invariant_ring` and `invariant_ring_random`, both by Agnes E. Heydtmann (`agnes@math.uni-sb.de`):

Bases of homogeneous invariants are generated successively and those are chosen as primary invariants that lower the dimension of the ideal generated by the previously found invariants (see paper "Generating a Noetherian Normalization of the Invariant Ring of a Finite Group" by Decker, Heydtmann, Schreyer (1997) to appear in JSC). In the non-modular case secondary invariants are calculated by finding a basis (in terms of monomials) of the basering modulo the primary invariants, mapping to invariants with the Reynolds operator and using those or their power products such that they are linearly independent modulo the primary invariants (see paper "Some Algorithms in Invariant Theory of Finite Groups" by Kemper and Steel (1997)). In the modular case they are generated according to "Generating Invariant Rings of Finite Groups over Arbitrary Fields" by Kemper (1996, to appear in JSC).

We calculate now an example from Sturmfels: "Algorithms in Invariant Theory 2.3.7":

```
LIB "finvar.lib";
ring R=0,(x,y,z),dp;
matrix A[3][3]=0,1,0,-1,0,0,0,0,-1;
// the group G is generated by A in Gl(3,Q);
print(A);
↦ 0, 1,0,
↦ -1,0,0,
↦ 0, 0,-1
// the fourth power of A is 1:
print(A*A*A*A); // the fourth power of A is 1
↦ 1,0,0,
↦ 0,1,0,
↦ 0,0,1
// Use the first method to compute the invariants of G:
matrix B(1..3);
B(1..3)=invariant_ring(A);
// SINGULAR returns 2 matrices, the first containing
// primary invariants and the second secondary
// invariants, i.e. module generators over a Noetherian
// normalization
// the third result are the irreducible secondary invariants
// if the Molien series was available
print(B(1));
↦ z2,x2+y2,x2y2
print(B(2));
↦ 1,xyz,x2z-y2z,x3y-xy3
print(B(3));
↦ xyz,x2z-y2z,x3y-xy3
// Use the second method,
// with using random numbers between -1 and 1:
B(1..3)=invariant_ring_random(A,1);
print(B(1..3));
↦ z2,x2+y2,x4+y4-z4
↦ 1,xyz,x2z-y2z,x3y-xy3
↦ xyz,x2z-y2z,x3y-xy3
```

A.19 Factorization

The factorization of polynomials is implemented in the C++ libraries Factory (written mainly by Ruediger Stobbe) and libfac (written by Michael Messollen) which are part of the SINGULAR system.

```
ring r = 0,(x,y),dp;
poly f = 9x16-18x13y2-9x12y3+9x10y4-18x11y2+36x8y4
       +18x7y5-18x5y6+9x6y4-18x3y6-9x2y7+9y8;
// = 9 * (x5-1y2)^2 * (x6-2x3y2-1x2y3+y4)
factorize(f);
↦ [1]:
```

```
↦    _[1]=9
↦    _[2]=x6-2x3y2-x2y3+y4
↦    _[3]=-x5+y2
↦ [2]:
↦    1,1,2
// returns factors and multiplicities,
// first factor is a constant.
poly g = (y4+x8)*(x2+y2);
factorize(g);
↦ [1]:
↦    _[1]=1
↦    _[2]=x2+y2
↦    _[3]=x8+y4
↦ [2]:
↦    1,1,1
// The same in characteristic 2:
ring r =2,(x,y),dp;
↦ // ** redefining r **
poly g = (y4+x8)*(x2+y2);
factorize(g);
↦ [1]:
↦    _[1]=1
↦    _[2]=x+y
↦    _[3]=x2+y
↦ [2]:
↦    1,2,4
```

A.20 Puiseux pairs

The Puiseux pairs of an irreducible and reduced curve singularity are its most important invariants. They can be computed from its Hamburger-Noether expansion. The library `hnoether.lib` written by Martin Lamm uses the algorithm of Antonio Campillo "Algebroid curves in positive characteristic" SLN 813, 1980. This algorithm has the advantage that it needs least possible field extensions and, moreover, works in any characteristic. This fact can be used to compute the invariants over a field of finite characteristic, say 32003, which will then be most probably be the same in characteristic 0.

We compute the Hamburger-Noether expansion of some plane curve singularities given by a polynomial f in two variables. This is a matrix which allows to compute the parametrization (up to a given order) and all numerical invariants like

- characteristic exponents
- puiseux pairs (of a complex model)
- degree of the conductor
- delta invariant
- generators of semigroup

Besides this, the library contains procedures to compute the Newton polygon of f, the squarefree part of f and a procedure to convert one set of invariants to another.

```
LIB "hnoether.lib";
// ======== The irreducible case ========
ring s = 0,(x,y),ds;
poly f = y4-2x3y2-4x5y+x6-x7;
list hn = develop(f);
↦ h(0) = 1
↦ a(1,2) = 1
↦ h(1) = 2
↦ a(2,2) = 1/4
↦ a(2,3) = -1/2
show(hn[1]);     // Hamburger-Noether matrix
↦ // matrix, 3x3
↦ 0,x,  0,
↦ 0,1,  x,
↦ 0,1/4,-1/2
displayHNE(hn);  // Hamburger-Noether development
↦ HNE[1]=-y+z(0)*z(1)
↦ HNE[2]=-x+z(1)^2+z(1)^2*z(2)
↦ HNE[3]=1/4*z(2)^2-1/2*z(2)^3
setring s;
displayInvariants(hn);
↦  characteristic exponents  : 4,6,7
↦  puiseux pairs             : (3,2)(7,2)
↦  degree of the conductor   : 16
↦  delta invariant           : 8
↦  generators of semigroup   : 4,6,13
↦  sequence of multiplicities: 4,2,2,1
// invariants(hn);  returns the invariants as list
                // partial parametrization of f
                // param takes the first variable as
param(hn);       // except the ring has >2 variables
↦ // ** Warning: result is exact up to order 5 in x and 7 in y !
↦ _[1]=1/16x4-3/16x5+1/4x7
↦ _[2]=1/64x6-5/64x7+3/32x8+1/16x9-1/8x10
ring extring=0,(x,y,t),ds;
poly f=x3+2xy2+y2;
list hn=develop(f,-1);
↦  Warning! You have defined too many variables!
↦  All variables except the first two will be ignored!
↦ h(0) = 1
↦ a(1,2) = -1
param(hn);       // partial parametrization of f
↦ // ** Warning: result is exact up to order 2 in x and 3 in y !
↦ _[1]=-t2
↦ _[2]=-t3
list hn1=develop(f,6);
↦  Warning! You have defined too many variables!
↦  All variables except the first two will be ignored!
↦ h(0) = 1
↦ a(1,2) = -1
↦ a(1,4) = 2
```

```
↦ a(1,6) = -4
param(hn1);     // a better parametrization
↦ // ** Warning: result is exact up to order 6 in x and 7 in y !
↦ _[1]=-t2+2t4-4t6
↦ _[2]=-t3+2t5-4t7
// instead or recomputing you may extend the development:
list hn2=extdevelop(hn,12);
↦ a(1,4) = 2
↦ a(1,6) = -4
↦ a(1,8) = 8
↦ a(1,10) = -16
↦ a(1,12) = 32
param(hn2);     // a still better parametrization
↦ // ** Warning: result is exact up to order 12 in x and 13 in y !
↦ _[1]=-t2+2t4-4t6+8t8-16t10+32t12
↦ _[2]=-t3+2t5-4t7+8t9-16t11+32t13
// ======== The reducible case ========
ring r = 0,(x,y),dp;
poly f=x11-2y2x8-y3x7-y2x6+y4x5+2y4x3+y5x2-y6;
// = (x5-1y2) * (x6-2x3y2-1x2y3+y4)
list hn=reddevelop(f);
↦ HNE of one branch found
↦ finite HNE found
↦   ~~~ result: 2  branch(es) successfully computed ~~~
show(hn[1][1]);     // Hamburger-Noether matrix of 1st brach
↦ // matrix, 3x3
↦ 0,x,0,
↦ 0,1,x,
↦ 0,1,-1
displayInvariants(hn);
↦  --- invariants of branch number 1 : ---
↦  characteristic exponents  : 4,6,7
↦  puiseux pairs             : (3,2)(7,2)
↦  degree of the conductor   : 16
↦  delta invariant           : 8
↦  generators of semigroup   : 4,6,13
↦  sequence of multiplicities: 4,2,2,1
↦
↦  --- invariants of branch number 2 : ---
↦  characteristic exponents  : 2,5
↦  puiseux pairs             : (5,2)
↦  degree of the conductor   : 4
↦  delta invariant           : 2
↦  generators of semigroup   : 2,5
↦  sequence of multiplicities: 2,2,1
↦
↦  -------------- intersection multiplicities : --------------
↦
↦ branch |    2
↦ -------+-----
↦      1  |   12
```

```
param(hn[2]);      // parametrization of 2nd branch
↦ _[1]=x2
↦ _[2]=x5
// extended parametrization of 1st
```

A.21 Primary decomposition

There are two algorithms implemented in SINGULAR which provide primary decomposition: `primdecGTZ`, based on Gianni/Trager/Zacharias (written by Gerhard Pfister) and `primdecSY`, based on Shimoyama/Yokoyama (written by Wolfram Decker and Hans Schoenemann).

The result of primdecGTZ is returned as a list of ideals, where the even indexed ideals form the prime ideal and the odd indexed ideals form the corresponding primary ideal.

The result of primdecSY is a list of list-tupels of ideals, the primary ideal and the corresponding prime ideal.

```
LIB "primdec.lib";
ring r = 0,(a,b,c,d,e,f),dp;
ideal i= f3, ef2, e2f, bcf-adf, de+cf, be+af, e3;
primdecGTZ(i);
↦ [1]:
↦    [1]:
↦       _[1]=f
↦       _[2]=e
↦    [2]:
↦       _[1]=f
↦       _[2]=e
↦ [2]:
↦    [1]:
↦       _[1]=f3
↦       _[2]=ef2
↦       _[3]=e2f
↦       _[4]=e3
↦       _[5]=de+cf
↦       _[6]=be+af
↦       _[7]=-bc+ad
↦    [2]:
↦       _[1]=f
↦       _[2]=e
↦       _[3]=-bc+ad
// We consider now the ideal J of the base space of the
// miniversal deformation of the cone over the rational
// normal curve computed in section *11* and compute
// its primary decomposition.
ring R = 0,(A,B,C,D),dp;
ideal J = CD, BD+D2, AD;
primdecGTZ(J);
↦ [1]:
↦    [1]:
```

```
↦       _[1]=D
↦    [2]:
↦       _[1]=D
↦ [2]:
↦    [1]:
↦       _[1]=C
↦       _[2]=B+D
↦       _[3]=A
↦    [2]:
↦       _[1]=C
↦       _[2]=B+D
↦       _[3]=A
// We see that there are two components which are both
// prime, even linear subspaces, one 3-dimensional,
// the other 1-dimensional.
// (This is Pinkhams example and was the first found
// surface singularity with two components of
// different dimensions)
//
// Let us now produce an embedded component in the last
// example, compute the minimal associated primes and
// the radical. We use the Characteristic set methods.
J = intersect(J,maxideal(3));
                  //shows that the maximal ideal is
primdecSY(J);     //embedded (takes a few seconds)
↦ [1]:
↦    [1]:
↦       _[1]=D
↦    [2]:
↦       _[1]=D
↦ [2]:
↦    [1]:
↦       _[1]=C
↦       _[2]=B+D
↦       _[3]=A
↦    [2]:
↦       _[1]=C
↦       _[2]=B+D
↦       _[3]=A
↦ [3]:
↦    [1]:
↦       _[1]=D2
↦       _[2]=C2
↦       _[3]=B2
↦       _[4]=AB
↦       _[5]=A2
↦       _[6]=BCD
↦       _[7]=ACD
↦    [2]:
↦       _[1]=D
↦       _[2]=C
```

```
↦       _[3]=B
↦       _[4]=A
minAssChar(J);
↦ [1]:
↦    _[1]=C
↦    _[2]=B+D
↦    _[3]=A
↦ [2]:
↦    _[1]=D
radical(J);
↦ _[1]=CD
↦ _[2]=BD+D2
↦ _[3]=AD
```

A.22 Normalization

The normalization will be computed for a reduced ring R/I. The result is a list of rings; ideals are always called KK in the rings of this list. The normalization of R/I is the product of the factor rings of the rings in the list divided out by the ideals KK.

```
LIB "normal.lib";
// ----- first example: rational quadruple point -----
ring R=32003,(x,y,z),wp(3,5,15);
ideal I=z*(y3-x5)+x10;
list pr=normal(I);
def S=pr[1];
setring S;
KK;
↦ KK[1]=T(2)*T(3)-T(1)*T(4)
↦ KK[2]=T(1)^7+T(1)^2*T(2)^3-T(1)^2*T(3)+T(2)*T(5)
↦ KK[3]=T(1)^4*T(2)^2+T(1)^2*T(5)-T(2)*T(4)
↦ KK[4]=T(1)^5*T(4)+T(1)^2*T(2)^2*T(5)+T(2)^3*T(4)-T(3)*T(4)+T(5)^2
↦ KK[5]=T(1)^6*T(3)+T(1)^2*T(2)^2*T(4)-T(1)*T(3)^2+T(4)*T(5)
↦ KK[6]=T(1)^4*T(2)*T(4)+T(1)*T(3)*T(5)-T(4)^2
// ----- second example: union of straight lines -----
ring R1=0,(x,y,z),dp;
ideal I=(x-y)*(x-z)*(y-z);
list qr=normal(I);
def S1=qr[1]; def S2=qr[2]; def S3=qr[3];
setring S1; KK;
↦ KK[1]=0
setring S2; KK;
↦ KK[1]=0
setring S3; KK;
↦ KK[1]=0
```

A.23 Kernel of module homomorphisms

Let A, B two matrices of size $m \times r$ and $m \times s$ over the ring R and consider the corresponding maps

$$R^r \xrightarrow{A} R^m \xleftarrow{B} R^s \ .$$

We want to compute the kernel of the map $R^r \xrightarrow{A} R^m \longrightarrow R^m/\mathrm{Im}(B)$. This can be done using the `modulo` command:

$$\texttt{modulo}(A,B) = \ker(R^r \xrightarrow{A} R^m/\mathrm{Im}(B)) \ .$$

```
ring r=0,(x,y,z),(c,dp);
matrix A[2][2]=x,y,z,1;
matrix B[2][2]=x2,y2,z2,xz;
modulo(A,B);
↦ _[1]=[yz2-x2,x2z-xz2]
↦ _[2]=[xyz-y2,-x2z+y2z]
↦ _[3]=[x2z-xy,xyz-yz2]
↦ _[4]=[x3-y2z]
```

A.24 Algebraic dependence

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

1. f_1, ..., f_r are algebraically dependent.
   Let $I = \langle Y_1 - f_1, \ldots, Y_r - f_r \rangle \subseteq K[x_1, \ldots, x_n, Y_1, \ldots, Y_r]$. Then $I \cap K[Y_1, \ldots, Y_r]$ are the algebraic relations between f_1, ..., f_r.
2. $g \in K[f_1, \ldots, f_r]$.
   $g \in K[f_1, \ldots, f_r]$ if and only if the normalform of g with respect to I and a blockordering with respect to $X = (x_1, \ldots, x_n)$ and $Y = (Y_1, \ldots, Y_r)$ with $X > Y$ is in $K[Y]$. Then $g = h(f_1, \ldots, f_r)$.

Both questions can be answered using the following procedure. If the second argument is zero, it checks for algebraic dependence and returns the ideal of relations between the generators of the given ideal. Otherwise it checks for subring membership and returns the normal form of the second argument with respect to the ideal I.

```
proc algebraicDep(ideal J, poly g)
{
  def R=basering;          // give a name to the basering
  int n=size(J);
  int k=nvars(R);
  int i;
  intvec v;

  // construction of the new ring

  v[n+k]=0;                // construct a weight vector
  for(i=1;i<=k;i++)
  {
```

```
    v[i]=1;
  }
  string orde="(a("+string(v)+"),dp);";
  string ri="ring Rhelp=("+charstr(R)+"),
                        ("+varstr(R)+",Y(1.."+string(n)+")),"+orde;
                          // ring definition as a string
  execute(ri);            // execution of the string

  // construction of the new ideal I=(J[1]-Y(1),...,J[n]-y(n))
  ideal I=imap(R,J);
  for(i=1;i<=n;i++)
  {
    I[i]=I[i]-var(k+i);
  }
  poly g=imap(R,g);
  if(g==0)
  {
    // construction of the ideal of relations by elimination
    poly el=var(1);
    for(i=2;i<=k;i++)
    {
      el=el*var(i);
    }
    ideal KK=eliminate(I,el);
    keepring(Rhelp);
    return(KK);
  }
  // reduction of g with respect to I
  ideal KK=reduce(g,std(I));
  keepring(Rhelp);
  return(KK);
}

// applications of the procedure
ring r=0,(x,y,z),dp;
ideal i=xz,yz;
algebraicDep(i,0);
↦ _[1]=0
setring r; kill Rhelp;
ideal j=xy+z2,z2+y2,x2y2-2xy3+y4;
algebraicDep(j,0);
↦ _[1]=Y(1)^2-2*Y(1)*Y(2)+Y(2)^2-Y(3)
setring r; kill Rhelp;
poly g=y2z2-xz;
algebraicDep(i,g);
↦ _[1]=Y(2)^2-Y(1)
setring r; kill Rhelp;
algebraicDep(j,g);
↦ _[1]=-z^4+z^2*Y(2)-x*z
```

A.25 Classification

Classification of isolated hypersurface singularities with respect to right equivalence is provided by the command `classify` of the library `classify.lib` written by Kai Krueger. The classification is done using the algorithm of Arnold. Before entering this algorithm, a first guess based on the Hilbert polynomial of the Milnor algebra is made.

```
LIB "classify.lib";
ring r=0,(x,y,z),ds;
poly p=singularity("E[6k+2]",2)[1];
p=p+z^2;
p;
↦ z2+x3+xy6+y8
// We received a E_14 singularity in normal form
// from the database of normal forms. Since only the residual
// part is saved in the database, we added z^2 to get an E_14
// of embedding dimension 3.
//
// Now we apply a coordinate change in order to deal with a
// singularity which is not in normal form:
map phi=r,x+y,y+z,x;
poly q=phi(p);
// Yes, q really looks ugly, now:
q;
↦ x2+x3+3x2y+3xy2+y3+xy6+y7+6xy5z+6y6z+15xy4z2+15y5z2+20xy3z3+20y4z3+15
↦ xy2z4+15y3z4+6xyz5+6y2z5+xz6+yz6+y8+8y7z+28y6z2+56y5z3+70y4z4+56y3z5+
↦ 28y2z6+8yz7+z8
// Classification
classify(q);
↦ About the singularity :
↦           Milnor number(f)   = 14
↦           Corank(f)          = 2
↦           Determinacy       <= 12
↦ Guessing type via Milnorcode:   E[6k+2]=E[14]
↦
↦ Computing normal form ...
↦ I have to apply the splitting lemma. This will take some time....:-)
↦    Arnold step number 9
↦ The singularity
↦    'x3-9/4x4+27/4x5-189/8x6+737/8x7+6x6y+15x5y2+20x4y3+15x3y4+6x2y5+x
↦ y6-24089/64x8-x7y+11/2x6y2+26x5y3+95/2x4y4+47x3y5+53/2x2y6+8xy7+y8+10
↦ 4535/64x9+27x8y+135/2x7y2+90x6y3+135/2x5y4+27x4y5+9/2x3y6-940383/128x
↦ 10-405/4x9y-2025/8x8y2-675/2x7y3-2025/8x6y4-405/4x5y5-135/8x4y6+43590
↦ 15/128x11+1701/4x10y+8505/8x9y2+2835/2x8y3+8505/8x7y4+1701/4x6y5+567/
↦ 8x5y6-82812341/512x12-15333/8x11y-76809/16x10y2-25735/4x9y3-78525/16x
↦ 8y4-16893/8x7y5-8799/16x6y6-198x5y7-495/4x4y8-55x3y9-33/2x2y10-3xy11-
↦ 1/4y12'
↦ is R-equivalent to E[14].
↦    Milnor number = 14
↦    modality      = 1
↦ z2+x3+xy6+y8
```

```
// The library also provides routines to determine the corank of q
// and its residual part without going through the whole
// classification algorithm.
corank(q);
↦ 2
morsesplit(q);
↦ y3-9/4y4+27/4y5-189/8y6+737/8y7+6y6z+15y5z2+20y4z3+15y3z4+6y2z5+yz6-2
↦ 4089/64y8-y7z+11/2y6z2+26y5z3+95/2y4z4+47y3z5+53/2y2z6+8yz7+z8+104535
↦ /64y9+27y8z+135/2y7z2+90y6z3+135/2y5z4+27y4z5+9/2y3z6-940383/128y10-4
↦ 05/4y9z-2025/8y8z2-675/2y7z3-2025/8y6z4-405/4y5z5-135/8y4z6+4359015/1
↦ 28y11+1701/4y10z+8505/8y9z2+2835/2y8z3+8505/8y7z4+1701/4y6z5+567/8y5z
↦ 6-82812341/512y12-15333/8y11z-76809/16y10z2-25735/4y9z3-78525/16y8z4-
↦ 16893/8y7z5-8799/16y6z6-198y5z7-495/4y4z8-55y3z9-33/2y2z10-3yz11-1/4z
↦ 12
```

A.26 Fast lexicographical GB

Compute Groebner basis in lexicographical ordering by using FGLM algorithm (`stdfglm`) and Hilbert driven Groebner (`stdhilb`).

The command `stdfglm` works only for zero dimensional ideals and returns a reduced Groebner basis.

For the ideal below, `stdfglm` is more than 100 times and `stdhilb` about 10 times faster than std.

```
ring r =32003,(a,b,c,d,e),lp;
ideal i=a+b+c+d, ab+bc+cd+ae+de, abc+bcd+abe+ade+cde,
        abc+abce+abde+acde+bcde, abcde-1;
int t=timer;
ideal j1=stdfglm(i);
timer-t;
↦ 0
size(j1);   // size (no. of polys) in computed GB
↦ 5
t=timer;
ideal j2=stdhilb(i);
timer-t;
↦ 1
size(j2);   // size (no. of polys) in computed GB
↦ 158
// usual Groebner basis computation for lex ordering
t=timer;
ideal j0 =std(i);
timer-t;
↦ 6
```

A.27 Parallelization with MPtcp links

In this example, we demonstrate how MPtcp links can be used to parallelize computations.

To compute a standard base for a zero-dimensional ideal in the lexicographical ordering, one of the two powerful routines `stdhilb` (see Section 5.1.108 [stdhilb], page 171)s and `stdfglm` (see Section 5.1.107 [stdfglm], page 170) should be used. However, one can not a priory predict which one the two commands is faster. This very much depends on the (input) example. Therefore, we use MPtcp links to let the two commands work on the problem independently and in parallel, so that the one which finishes first delivers the result.

The example we use is the so-called "omndi example". See *Tim Wichmann; Der FGLM-Algorithmus: verallgemeinert und implementiert in Singular; Diplomarbeit Fachbereich Mathematik, Universitaet Kaiserslautern; 1997* for more details.

```
ring r=0,(a,b,c,u,v,w,x,y,z),lp;
ideal i=a+c+v+2x-1, ab+cu+2vw+2xy+2xz-2/3,  ab2+cu2+2vw2+2xy2+2xz2-2/5,
ab3+cu3+2vw3+2xy3+2xz3-2/7, ab4+cu4+2vw4+2xy4+2xz4-2/9, vw2+2xyz-1/9,
vw4+2xy2z2-1/25, vw3+xyz2+xy2z-1/15, vw4+xyz3+xy3z-1/21;

link l_hilb,l_fglm = "MPtcp:fork","MPtcp:fork";         // 1.

open(l_fglm); open(l_hilb);

write(l_hilb, quote(system("pid")));                    // 2.
write(l_fglm, quote(system("pid")));
int pid_hilb,pid_fglm = read(l_hilb),read(l_fglm);

write(l_hilb, quote(stdhilb(i)));                       // 3.
write(l_fglm, quote(stdfglm(eval(i))));

while ((! status(l_hilb, "read", "ready", 1)) &&        // 4.
     (! status(l_fglm, "read", "ready", 1))) {}

if (status(l_hilb, "read", "ready"))
{
"stdhilb won !!!!"; size(read(l_hilb));
close(l_hilb); pid_fglm = system("sh","kill "+string(pid_fglm));
}
else                                                    // 5.
{
"stdfglm won !!!!"; size(read(l_fglm));
close(l_fglm); pid_hilb = system("sh","kill "+string(pid_hilb));
}
↦ stdfglm won !!!!
↦ 9
```

Some explainatory remarks are in order:

1. Instead of using links of the type `MPtcp:fork`, we alternatively could use `MPtcp:launch` links such that the two "competing" SINGULAR processes run on different machines. This has the

advantage of "true" parallel computing since no resource sharing is involved (as it usually is with forked processes).

2. Unfortunately, MPtcp links do not offer means to (asynchronously) interrupt or kill an attached (i.e., launched or forked) process. Therefore, we explicitely need to get the process id numbers of the competing SINGULAR processes, so that we can later "kill" the looser.
3. Notice how quoting is used in order to prevent local evaluation (i.e., local computation of results). Since we "forked" the two competing processes, the identifier `i` is defined and has identical values in the two child processes. Therefore, the innermost `eval` can be ommited (as is done for the `l_hilb` link), and only the identifier `i` needs to be communicated to the children. However, when `MPtcp:launch` links are used, the inner evaluation must be applied so that actual values, and not the identifiers are communicated (as is done for the `l_fglm` link).
4. We go into a "sleepy" loop and wait until one of the two children finished the computation. That is, the current process checks appr. once a per second the status of one of the connecting links, and sleeps (i.e., suspends its execution) in the intermediate time.
5. The child which won delivers the result and is terminated with the usual `close` command. The other child which is still computing needs to be terminated by an explicit (i.e., system) kill command, since it can not be terminated through the link while it is still computing.

Appendix B Polynomial data

B.1 Representation of mathematical objects

SINGULAR distinguishes between objects which do not belong to a ring and those which belong to a specific ring (see Section 3.2 [Rings and orderings], page 20). We comment only on the latter ones.

Internally all ring-dependent objects are polynomials or structures built from polynomials (and some additional information). Note that SINGULAR stores (and hence prints) a polynomial automatically w.r.t. the monomial ordering.

Hence, in order to define such an object in SINGULAR, one has to give a list of polynomials in a specific format.

For ideals resp. matrices this is straight forward: The user gives a list of polynomials which generate the ideal resp. which are the entries of the matrix. (The number of rows and columns have to be given when creating the matrix.)

A vector in SINGULAR is always an element of a free module over the basering. It is given as a list of polynomials in one of the following formats $[f_1, ..., f_n]$ or $f_1 * gen(1) + ... + f_n * gen(n)$, where $gen(i)$ denotes the i-th canonical generator of a free module (with 1 at place i and 0 everywhere else) Both forms are equivalent. A vector is internally represented in the second form with the $gen(i)$ being "special" ring variables, ordered accordingly to the monomial ordering. Therefore, the form $[f_1, ..., f_n]$ is given as output only if the monomial ordering gives priority to the component, i.e., is of the form `(c,...)` (see Section B.2.3 [Module orderings], page 242). However, the procedure `show` from the library `inout.lib` displays always the bracket form.

A vector $v = [f_1, ..., f_n]$ should always be considered as a column vector in a free module of rank equal to nrows(v) where nrows(v) is equal to the maximal index r such that $f_r \neq 0$ This is due to the fact, that internally v is a polynomial in a sparse representation, i.e. $f_i * gen(i)$ is not stored if $f_i = 0$ (for reasons of efficiency), hence the last 0-entries of v are lost. Only more complex structures are able to keep the rank.

A module M in SINGULAR is given by a list of vectors $v_1, ..., v_k$ which generate the module as a submodule of the free module of rank equal to nrows(M) which is the maximum of nrows(v_i).

If one wants to create a module with a larger rank than given by its generators, one has to use the command `attrib(m,"rank",r)` (see Section 5.1.1 [attrib], page 102, Section 5.1.75 [nrows], page 149) or define a matrix first, then converting it into a module. Modules in SINGULAR are almost the same as matrices, they may be considered as sparse representations of matrices. A module of a matrix is generated by the columns of the matrix and a matrix of a module has as columns the generators of the module. These conversions preserve the rank and the numbers of generators resp. the numbers of rows and columns.

By the above remarks it may appear that SINGULAR is only able to handle submodules of a free module. This is not true, SINGULAR can compute with any finitely generated module over the basering R. Such a module, say N, is not represented by its generators but by its (generators and) relations. This means that N=R^n/M where n is the number of generator of N and M$\subseteq R^n$ the

module of relations, and N is given by M. In other words, defining in SINGULAR a module M as a submodule of a free module R^n can also be considered as the definition of N=R^n/M.

Most functions in SINGULAR, when applied to M, really deal with M, but others deal with N=R^n/M. For example, `std(M)` computes a standard basis of M (and thus gives another representation of N as N=R^n/std(M)). But `dim(M)` resp. `vdim(M)` returns dim(R^n/M) resp. $\dim_k(R^n/M)$, if M is given by a standard basis. The function `syz(M)` returns the first syzygy of M, i.e., the module of relations of the given generators of M which is equal to the second syzygy module of N. Refer to the description of each function in Section 5.1 [Functions], page 102 to get information which module the functions deals with.

The numbering in `res` and other commands for computing resolutions refer to a resolution of N=R^n/M (see Section 5.1.96 [res], page 162; Section C.3 [Syzygies and resolutions], page 247).

Since any ring in SINGULAR is defined over a ground field, it is possible to compute in any field which is a valid ground field in SINGULAR. For doing so, one has to define a ring with the desired ground field and at least one variable. The elements of the field are of type number, but may also be considered as polynomials (of degree 0). Large computations should be faster if the elements of the field are defined as numbers.

The above remarks do also apply to quotient rings. Polynomial data are stored internally in the same manner, the only difference is that this polynomial representation is in general not unique. `reduce(f,std(0))` computes a normal form of a polynomial f in a quotient ring (cf. Section 5.1.94 [reduce], page 161).

B.2 Monomial orderings

A monomial ordering (term ordering) on $K[x_1,\ldots,x_n]$ is a total ordering $<$ on the set of monomials (power products) $\{x^\alpha | \alpha \in \mathbf{N^n}\}$ which is compatible with the natural semigroup structure, i.e. $x^\alpha < x^\beta$ implies $x^\gamma x^\alpha < x^\gamma x^\beta$ for any $\gamma \in \mathbf{N^n}$. We do not require $<$ to be a wellordering. See the literature cited in Section C.5 [References], page 248.

It is known that any monomial ordering can be represented by a matrix M in GL(n,R), but, of course, only integer coefficients are of relevance in practice.

Global orderings are wellorderings (i.e. $1 < x_i$ for each variable x_i), local orderings satisfy $1 > x_i$ for each variable. If some variables are ordered globally and others locally we call it a mixed ordering. Local or mixed orderings are not wellorderings.

If K is the groundfield, $x = (x_1,\ldots,x_n)$ the variables and $<$ a monomial ordering, then Loc K[x] denotes the localization of $K[x]$ with respect to the multiplicatively closed set $\{1 + g \mid g = 0$ or $g \in K[x]\backslash\{0\}$ and $L(g) < 1\}$. $L(g)$ denotes the leading monomial of g, i.e. the biggest monomial of g with respect to $<$. The result of any computation which uses standard basis computations has to be interpreted in Loc K[x]. SINGULAR offers the monomial orderings described in the following sections, a definition of a ring includes the definition of its monomial ordering, see Section 3.2 [Rings and orderings], page 20. in an effective way:

B.2.1 Global orderings

For all these orderings: Loc K[x] = K[x]

lp: lexicographical ordering.
$x^\alpha < x^\beta \Leftrightarrow \exists\, 1 \le i \le n : \alpha_1 = \beta_1, \ldots, \alpha_{i-1} = \beta_{i-1}, \alpha_i < \beta_i$.

dp: degree reverse lexicographical ordering.
$x^\alpha < x^\beta \Leftrightarrow \deg(x^\alpha) < \deg(x^\beta)$, where $\deg(x^\alpha) = \alpha_1 + \cdots + \alpha_n$, or
$\deg(x^\alpha) = \deg(x^\beta)$ and $\exists\, 1 \le i \le n :$
$\alpha_n = \beta_n, \ldots, \alpha_{i+1} = \beta_{i+1}, \alpha_i > \beta_i$.

Dp: degree lexicographical ordering.
$x^\alpha < x^\beta \Leftrightarrow \deg(x^\alpha) < \deg(x^\beta)$, where $\deg(x^\alpha) = \alpha_1 + \cdots + \alpha_n$, or
$\deg(x^\alpha) = \deg(x^\beta)$ and $\exists\, 1 \le i \le n :$
$\alpha_1 = \beta_1, \ldots, \alpha_{i-1} = \beta_{i-1}, \alpha_i < \beta_i$.

wp: weighted reverse lexicographical ordering.
`wp`$(w_1, \ldots, w_n)$, w_i positive integers, is defined as `dp` but with $\deg(x^\alpha) = w_1\alpha_1 + \cdots + w_n\alpha_n$.

Wp: weighted lexicographical ordering.
`Wp`$(w_1, \ldots, w_n)$, w_i positive integers, is defined as `Dp` but with $\deg(x^\alpha) = w_1\alpha_1 + \cdots + w_n\alpha_n$.

B.2.2 Local orderings

For ls, ds, Ds and, if the weights are positive integers, also for ws and Ws, we have Loc K$[x]$ = K$[x]_{(x)}$, the localization of K[x] at the maximal ideal $(x_1, ..., x_n)$.

ls: negative lexicographical ordering.
$x^\alpha < x^\beta \Leftrightarrow \exists\, 1 \le i \le n : \alpha_1 = \beta_1, \ldots, \alpha_{i-1} = \beta_{i-1}, \alpha_i > \beta_i$.

ds: negative degree reverse lexicographical ordering.
$x^\alpha < x^\beta \Leftrightarrow \deg(x^\alpha) > \deg(x^\beta)$, where $\deg(x^\alpha) = \alpha_1 + \cdots + \alpha_n$, or
$\deg(x^\alpha) = \deg(x^\beta)$ and $\exists\, 1 \le i \le n :$
$\alpha_n = \beta_n, \ldots, \alpha_{i+1} = \beta_{i+1}, \alpha_i > \beta_i$.

Ds: negative degree lexicographical ordering.
$x^\alpha < x^\beta \Leftrightarrow \deg(x^\alpha) > \deg(x^\beta)$, where $\deg(x^\alpha) = \alpha_1 + \cdots + \alpha_n$, or
$\deg(x^\alpha) = \deg(x^\beta)$ and $\exists\, 1 \le i \le n :$
$\alpha_1 = \beta_1, \ldots, \alpha_{i-1} = \beta_{i-1}, \alpha_i < \beta_i$.

ws: (general) weighted reverse lexicographical ordering.
`ws`$(w_1, \ldots, w_n)$, w_1 a nonzero integer, $w_2, \ldots, w_n$ any integer (including 0), is defined as `ds` but with $\deg(x^\alpha) = w_1\alpha_1 + \cdots + w_n\alpha_n$.

Ws: (general) weighted lexicographical ordering.
`Ws`$(w_1, \ldots, w_n)$, w_1 a nonzero integer, $w_2, \ldots, w_n$ any integer (including 0), is defined as `Ds` but with $\deg(x^\alpha) = w_1\alpha_1 + \cdots + w_n\alpha_n$.

B.2.3 Module orderings

SINGULAR offers also orderings on the set of "monomials" $\{x^a e_i | a \in N^n, 1 \leq i \leq r\}$ on Loc K[x]^r = Loc K[x]$e_1 + \ldots +$Loc K[x]e_r, where $e_1, \ldots, e_r$ denote the canonical generators of Loc K[x]^r, the r-fold direct sum of Loc K[x]. (The function `gen(i)` yields e_i).

We have two possibilities, either to give priority to the component of a vector of Loc K[x]^r or (which is the default in SINGULAR) to give priority to the coefficients. The orderings `(<,c)` and `(<,C)` give priority to the coefficients; `(c,<)` and `(C,<)` give priority to the components.
Let `<` be any of the monomial orderings of Loc K[x] as above.

(<,C): $<_m = (<, C)$ denotes the module ordering (giving priority to the coefficients):
$x^\alpha e_i <_m x^\beta e_j \Leftrightarrow x^\alpha < x^\beta$,
or $x^\alpha = x^\beta$ and $i < j$.

Example:

```
ring r = 0, (x,y,z), ds;
// the same as ring r = 0, (x,y,z), (ds, C);
[x+y2,z3+xy];
↦ x*gen(1)+xy*gen(2)+y2*gen(1)+z3*gen(2)
[x,x,x];
↦ x*gen(3)+x*gen(2)+x*gen(1)
```

(C,<): $<_m = (C, <)$ denotes the module ordering (giving priority to the component):
$x^\alpha e_i <_m x^\beta e_j \Leftrightarrow i < j$,
or $i = j$ and $x^\alpha < x^\beta$.

Example:

```
ring r = 0, (x,y,z), (C,lp);
[x+y2,z3+xy];
↦ xy*gen(2)+z3*gen(2)+x*gen(1)+y2*gen(1)
[x,x,x];
↦ x*gen(3)+x*gen(2)+x*gen(1)
```

(<,c): $<_m = (<, c)$ denotes the module ordering (giving priority to the coefficients):
$x^\alpha e_i <_m x^\beta e_j \Leftrightarrow x^\alpha < x^\beta$,
or $x^\alpha = x^\beta$ and $i > j$.

Example:

```
ring r = 0, (x,y,z), (lp,c);
[x+y2,z3+xy];
↦ xy*gen(2)+x*gen(1)+y2*gen(1)+z3*gen(2)
[x,x,x];
↦ x*gen(1)+x*gen(2)+x*gen(3)
```

(c,<): $<_m = (c, <)$ denotes the module ordering (giving priority to the component):
$x^\alpha e_i <_m x^\beta e_j \Leftrightarrow i > j$,
or $i = j$ and $x^\alpha < x^\beta$.

Example:

```
ring r = 0, (x,y,z), (c,lp);
[x+y2,z3+xy];
↦ [x+y2,xy+z3]
[x,x,x];
↦ [x,x,x]
```

The output of a vector v in $K[x]^r$ with components $v_1, \ldots, v_r$ has the format $v_1 * gen(1) + \ldots + v_r * gen(r)$ unless the ordering starts with `c`. In this case a vector is written as $[v_1, \ldots, v_r]$. In all cases SINGULAR can read the input in both formats.

B.2.4 Matrix orderings

Let M be an invertible n x n matrix with integer coefficients and $M_1, \ldots, M_n$ the rows of M.

The M-ordering `<` is the following: $x^a < x^b \Leftrightarrow$ there exists an $i : 1 <= i <= n : M_1 * a = M_1 * b, \ldots, M_{i-1} * a = M_{i-1} * b$ and $M_i * a < M_i * b$.

Thus, $x^a < x^b$ if and only if $M * a$ is smaller than $M * b$ with respect to the lexicographical ordering.

The following matrices represent (for 3 variables) the global and local orderings defined above (note that the matrix is not uniquely determined by the ordering):

lp: $\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$ dp: $\begin{pmatrix} 1 & 1 & 1 \\ 0 & 0 & -1 \\ 0 & -1 & 0 \end{pmatrix}$ Dp: $\begin{pmatrix} 1 & 1 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix}$

wp(1,2,3): $\begin{pmatrix} 1 & 2 & 3 \\ 0 & 0 & -1 \\ 0 & -1 & 0 \end{pmatrix}$ Wp(1,2,3): $\begin{pmatrix} 1 & 2 & 3 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix}$

ls: $\begin{pmatrix} -1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & -1 \end{pmatrix}$ ds: $\begin{pmatrix} -1 & -1 & -1 \\ 0 & 0 & -1 \\ 0 & -1 & 0 \end{pmatrix}$ Ds: $\begin{pmatrix} -1 & -1 & -1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix}$

ws(1,2,3): $\begin{pmatrix} -1 & -2 & -3 \\ 0 & 0 & -1 \\ 0 & -1 & 0 \end{pmatrix}$ Ws(1,2,3): $\begin{pmatrix} -1 & -2 & -3 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix}$

Product orderings (see next section) represented by a matrix:

(dp(3), wp(1,2,3)):

$$\begin{pmatrix} 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & -1 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 2 & 3 \\ 0 & 0 & 0 & 0 & 0 & -1 \\ 0 & 0 & 0 & 0 & -1 & 0 \end{pmatrix}$$

(Dp(3), ds(3)):

$$\begin{pmatrix} 1 & 1 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & -1 & -1 & -1 \\ 0 & 0 & 0 & 0 & 0 & -1 \\ 0 & 0 & 0 & 0 & -1 & 0 \end{pmatrix}$$

Orderings with extra weight vector (see below) represented by a matrix:

(dp(3), a(1,2,3),dp(3)):

$$\begin{pmatrix} 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & -1 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 2 & 3 \\ 0 & 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & -1 \\ 0 & 0 & 0 & 0 & -1 & 0 \end{pmatrix}$$

(a(1,2,3,4,5),Dp(3), ds(3)):

$$\begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 0 \\ 1 & 1 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & -1 & -1 & -1 \\ 0 & 0 & 0 & 0 & 0 & -1 \\ 0 & 0 & 0 & 0 & -1 & 0 \end{pmatrix}$$

Example:

```
ring r = 0, (x,y,z), M(1, 0, 0,
                       0, 1, 0,
                       0, 0, 1);
```

which may also be written as:

```
intmat m[3][3]=1, 0, 0, 0, 1, 0, 0, 0, 1;
m;
↦ 1,0,0,
↦ 0,1,0,
↦ 0,0,1
ring r = 0, (x,y,z), M(m);
r;
↦ //   characteristic : 0
↦ //   number of vars : 3
↦ //        block   1 : ordering M
↦ //                  : names    x y z
↦ //                  : weights  1 0 0
↦ //                  : weights  0 1 0
↦ //                  : weights  0 0 1
↦ //        block   2 : ordering C
```

If the ring has n variables and the matrix contains less than n x n entries an error message is given, if there are more entries, the last ones are ignored.

WARNING: SINGULAR does not check whether the matrix has full rank. In such a case some computations might not terminate, others might give a nonsense result.

Having these matrix orderings SINGULAR can compute standard bases for any monomial ordering which is compatible with the natural semigroup structure. In practice the global and local orderings together with block orderings should be sufficient in most cases. These orderings are faster than the corresponding matrix orderings, since evaluating a matrix product is time consuming.

B.2.5 Product orderings

Let $x = (x_1, \ldots, x_n) = x(1..n)$ and $y = (y_1, \ldots, y_m) = y(1..m)$ be two ordered sets of variables, $<_1$ a monomial ordering on $K[x]$ and $<_2$ a monomial ordering on $K[y]$. The product ordering (or block ordering) $< := (<_1, <_2)$ on $K[x, y]$ is the following:

$$x^a y^b < x^A y^B \Leftrightarrow x^a <_1 x^A \text{ or } x^a = x^A \text{ and } y^b <_2 y^B.$$

Inductively one defines the product ordering of more than two monomial orderings.

In SINGULAR, any of the above global orderings, local orderings or matrix ordering may be combined (in an arbitrary manner and length) to a product ordering. E.g. `(lp(3), M(1, 2, 3, 1, 1, 1, 1, 0, 0), ds(4), ws(1,2,3))` defines: `lp` on the first 3 variables, the matrix ordering `M(1, 2, 3, 1, 1, 1, 1, 0, 0)` on the next 3 variables, `ds` on the next 4 variables and `ws(1,2,3)` on the last 3 variables.

B.2.6 Extra weight vector

a: `a`$(w_1, \ldots, w_n)$, $w_1, \ldots, w_n$ any integer (including 0), defines $\deg(x^\alpha) = w_1\alpha_1 + \cdots + w_n\alpha_n$.

$$x^\alpha < x^\beta \Leftarrow \deg(x^\alpha) < \deg(x^\beta),$$
$$x^\alpha > x^\beta \Leftarrow \deg(x^\alpha) > \deg(x^\beta)$$

An extra weight vector does not define a monomial ordering by itself: it can only be used in combination with other orderings to insert an extra line of weights into the ordering matrix.
Example:

```
ring r = 0, (x,y,z), (a(1,2,3),
                      wp(4,5,2));
ring s = 0, (x,y,z), (a(1,2,3),dp);
ring q=  0, (a,b,c,d),(lp(1),a(1,2,3),ds);
```

Appendix C Mathematical background

This chapter introduces some of the mathematical notions and definitions used throughout the manual. It is mostly a collection of the most prominent definitions and properties and refer for details to some articles or text books.

C.1 Standard bases

Definition

Let $I \subseteq \mathrm{Loc}_<\mathrm{K}[\underline{x}]$ be an ideal. $\mathbf{L}(I)$ denotes the ideal of $\mathrm{Loc}_<\mathrm{K}[\underline{x}]$ generated by $\{L(f)|f \in I\}$. $f_1, \ldots, f_s \in I$ is called a **standard basis** of I if $\{L(f_1), \ldots, L(f_s)\}$ generates the ideal $L(I) \subset \mathrm{Loc}_<\mathrm{K}[\underline{x}]$.

Properties

Normal form:
: A function $NF : K[\underline{x}]^r \times \{G|G\ standardbasis\} \to K[\underline{x}]^r, (p, G) \mapsto NF(p|G)$, is called a **normal form** if for any $p \in K[\underline{x}]^r$ and any G the following holds: if $NF(p|G) \neq 0$ then $L(g) \not| L(NF(p|G))$ for all $g \in G$. $NF(g|G)$ is called a **normal form of p with respect to G** (such a function is not unique).

ideal membership:
: $f \in I$ iff $NF(f, std(I)) = 0$ for $I \subseteq R$ resp. $I \subseteq R^r$.

Hilbert function:
: Let $I \subseteq K[\underline{x}]^r$ be homogeneous, then the Hilbert function H_I of the ideal I and the Hilbert function $H_L(I)$ of the leading ideal $L(I)$ coincides. $H_I = H_{L(I)}$.

C.2 Hilbert function

Let M $= \bigoplus M_i$ be a graded module over $K[x_1, ..., x_n]$. The **Hilbert function of M** H_M is defined by

$$H_M(t) = dim_K M_t.$$

The **Hilbert-Poincare series of M** is the power series

$$HP_M(t) = \sum_{i=-\infty}^{\infty} H_M(i)t^i = \sum_{i=-\infty}^{\infty} dim_K M_i t^i$$

It turns out that $HP_M(t)$ can be written in two useful ways:

$$HP_M(t) = \frac{Q(t)}{(1-t)^n} = \frac{P(t)}{(1-t)^{dim(M)}}$$

where Q(t) and P(t) are polynomials from Z[t]. Q(t) is called **first Hilbert series**, and P(t) the **second Hilbert series**.

If $P(t) = \sum_{k=0}^{N} a_k t^k$ then $H_M(s) = \sum_{k=1}^{N} a_k \binom{d+n-k-1}{d-1}$ (the **Hilbert polynomial**) for $s \geq N$.

C.3 Syzygies and resolutions

Syzygies

Let $I = (g_1, ..., g_q) \subseteq K[x]^r$.
The **module of syzygies syz(I)** is $ker(K[x]^q \to K[x]^r, \sum_i w_i * e_i \mapsto \sum_i w_i * g_i)$.

Free resolutions

Let $I = (g_1, ..., g_s) \subseteq K[\underline{x}]^r$ and $M = K[\underline{x}]^r/I$. A **free resolution of** M is a long exact sequence

$$... \longrightarrow F_2 \xrightarrow{A_2} F_1 \xrightarrow{A_1} F_0 \longrightarrow M \longrightarrow 0,$$

where the columns of the matrix A_1 are generators of I. The length of the sequence is at most n, where n is the number of variables in the polynomial resp. power series ring. Free resolutions over other rings may be infinite. Considered as modules, A_1 is a set of generators of the input, A_2 consists of a set of generators of the first szyzgy module of A_1, etc.

Betti numbers

The graded **Betti number** $b_{i,j}$ of R^n/M (M a homogeneous submodule of R^n) is the minimal number of generators in degree i+j of the j-th syzygy module (= module of relations) of R^n/M (the 0-th (resp.1-st) syzygy module of R^n/M is R^n (resp. M)).

Regularity

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.

C.4 Characteristic sets

Let $<$ be the lexicographical ordering $x_1 < ... < x_n$ on $R = K[x_1, ..., x_n]$. For $f \in R$ let lvar(f) (the leading variable of f) be the largest variable in lead(f) (the leading term of f with respect to $<$), i.e. if $f = a_k(x_1, ..., x_{k-1})x_k^s + ... + a_0(x_1, ..., x_{k-1})$ for some $k \leq n$ then $lvar(f) = x_k$, moreover let $ini(f) := a_k(x_1, ..., x_{k-1})$.

A set $T = \{f_1, ..., f_r\} \subset R$ is called triangular if $lvar(f_1) < ... < lvar(f_r)$. The pseudoremainder $r = prem(g, f)$ of g with respect to f is defined by $ini(f)^a * g = q * f + r$ with the property $deg_{lvar(f)}(r) < deg_{lvar(f)}(f)$, a minimal.

(T,U) is called a triangular system, if U$\subset$T and if T is a triangular set such that $ini(T)$ does not vanish on $Zero(T) \setminus Zero(U)(=: Zero(T \setminus U))$.

T is called irreducible if for every i there are no d_i, f_i', f_i'' with the property:

$$lvar(d_i) < lvar(f_i)$$

$$lvar(f_i') = lvar(f_i'') = lvar(f_i)$$

$$0 \notin prem(\{d_i, ini(f_i'), ini(f_i'')\}, \{f_1, ..., f_{i-1}\})$$

such that $prem(d_i * f_i - f_i' * f_i'', \{f_1, ..., f_{i-1}\}) = 0$.

(T,U) is irreducible if T is irreducible.

Let $G = \{g_1, ..., g_s\}$ then there are irreducible triangular sets $T_1, ..., T_l$ such that $Zero(G) = \bigcup_{i=1..l}(Zero(T_i \setminus I_i))$ where $I_i = \{ini(f), f \in T_i\}$.

The set $\{T_1, ..., T_l\}$ is called a **irreducible characteristic series** of the ideal (G).

Example:

```
ring R= 0,(x,y,z,u),dp;
ideal i=-3zu+y2-2x+2,
        -3x2u-4yz-6xz+2y2+3xy,
        -3z2u-xu+y2z+y;
print(char_series(i));
↦ _[1,1],3x2z-y2+2yz,3x2u-3xy-2y2+2yu,
↦ x,     -y+2z,      -2y2+3yu-4
```

C.5 References

The Centre for Computer Algebra Kaiserslautern publishes a series of preprint which are electronically available at `http://www.mathematik.uni-kl.de/~zca/Reports_on_ca`. Other sources to check are `http://symbolicnet.mcs.kent.edu/`, `http://www.can.nl/`,... and the following book list:

Text books on computational algebraic geometry

- Adams, W.; Loustaunau, P.: An Introduction to Gröbner Bases. Providence, RI, AMS, 1996
- Becker, T.; Weisspfenning, V.: Gröbner Bases - A Computational Approach to Commutative Algebra. Springer, 1993
- Cohen, H.: A Course in Computational Algebraic Number Theory, Springer, 1995
- Cox, D.; Little, J.; O'Shea, D.: Ideals, Varieties and Algorithms. Springer, 1996
- Eisenbud, D.: Commutative Algebra with a View Toward Algebraic Geometry. Springer, 1995
- Mishra, B.: Algorithmic Algebra, Texts and Monographs in Computer Science. Springer, 1993
- Sturmfels, B.: Algorithms in Invariant Theory. Springer 1993
- Vasconcelos, W.: Computational Methods in Commutative Algebra and Algebraic Geometry. Springer, 1998

Descriptions of algorithms

- Bareiss, E.: Sylvester's identity and multistep integer-preserving Gaussian elimination. Math. Comp. 22 (1968), 565-578
- Campillo, A.: Algebroid curves in positive characteristic. SLN 813, 1980
- Chou, S.: Mechanical Geometry Theorem Proving. D.Reidel Publishing Company, 1988
- Decker, W.; G.-M. Greuel, G.-M.; Pfister, G.: Primary decomposition: algorithms and comparisons. Preprint, Univ. Kaiserslautern, 1998. To appear in: Greuel, G.-M.; Matzat, B. H.; Hiss, G. (Eds.), Algorithmic Algebra and Number Theory. Springer Verlag, Heidelberg, 1998
- Decker, W.; G.-M. Greuel, G.-M.; de Jong, T.; Pfister, G.: The normalisation: a new algorithm, implementation and comparisons. Preprint, Univ. Kaiserslautern, 1998
- Decker, W.; Heydtmann, A.; Schreyer, F. O.: Generating a Noetherian Normalization of the Invariant Ring of a Finite Group, 1997, to appear in Journal of Symbolic Computation
- Faugère, J. C.; Gianni, P.; Lazard, D.; Mora, T.: Efficient computation of zero-dimensional Gröbner bases by change of ordering. Journal of Symbolic Computation, 1989
- Gräbe, H.-G.: On factorized Gröbner bases, Univ. Leibzig, Inst. f"ur Informatik, 1994
- Grassmann, H.; Greuel, G.-M.; Martin, B.; Neumann, W.; Pfister, G.; Pohl, W.; Schönemann, H.; Siebert, T.: On an implementation of standard bases and syzygies in SINGULAR. Proceedings of the Workshop Computational Methods in Lie theory in AAECC (1995)
- Greuel, G.-M.; Pfister, G.: Advances and improvements in the theory of standard bases and syzygies. Arch. d. Math. 63(1995)
- Kemper; Generating Invariant Rings of Finite Groups over Arbitrary Fields. 1996, to appear in Journal of Symbolic Computation
- Kemper and Steel: Some Algorithms in Invariant Theory of Finite Groups. 1997
- Schönemann, H.: Algorithms in SINGULAR, Reports on Computer Algebra 2(1996), Kaiserslautern
- Siebert, T.: On strategies and implementations for computations of free resolutions. Reports on Computer Algebra 8(1996), Kaiserslautern
- Wang, D.: Characteristic Sets and Zero Structure of Polynomial Sets. Lecture Notes, RISC Linz, 1989

Appendix D SINGULAR libraries

SINGULAR comes with a set of standard libraries. Their content is described in the following subsections.

D.1 standard_lib

The library `standard.lib` provides extensions to the set of built-in commands and is automatically loaded during the start of SINGULAR. It contains:

`groebner` see Section 5.1.37 [groebner], page 124.

`stdfglm` see Section 5.1.107 [stdfglm], page 170.

`stdhilb` see Section 5.1.108 [stdhilb], page 171.

D.2 all_lib

```
LIBRARY:  all.lib   Load all libraries

classify.lib  PROCEDURES FOR THE ARNOLD-CLASSIFIER OF SINGULARITIES
deform.lib    PROCEDURES FOR COMPUTING MINIVERSAL DEFORMATION
elim.lib      PROCEDURES FOR ELIMINATION, SATURATION AND BLOWING UP
finvar.lib    PROCEDURES TO CALCULATE INVARIANT RINGS & MORE
general.lib   PROCEDURES OF GENERAL TYPE
graphics.lib  PROCEDURES TO DRAW WITH MATHEMATICA
hnoether.lib  PROCEDURES FOR THE HAMBURGER-NOETHER-DEVELOPMENT
homolog.lib   PROCEDURES FOR HOMOLOGICAL ALGEBRA
inout.lib     PROCEDURES FOR MANIPULATING IN- AND OUTPUT
invar.lib     PROCEDURES FOR COMPUTING INVARIANTS OF (C,+)-ACTIONS
matrix.lib    PROCEDURES FOR MATRIX OPERATIONS
normal.lib    PROCEDURES FOR COMPUTING THE NORMALIZATION
poly.lib      PROCEDURES FOR MANIPULATING POLYS, IDEALS, MODULES
presolve.lib  PROCEDURES FOR PRE-SOLVING POLYNOMIAL EQUATIONS
primdec.lib   PROCEDURES FOR PRIMARY DECOMPOSITION
primitiv.lib  PROCEDURES FOR FINDING A PRIMITIVE ELEMENT
random.lib    PROCEDURES OF RANDOM MATRIX AND POLY OPERATIONS
ring.lib      PROCEDURES FOR MANIPULATING RINGS AND MAPS
sing.lib      PROCEDURES FOR SINGULARITIES
standard.lib  PROCEDURES WHICH ARE ALWAYS LOADED AT START-UP
latex.lib     PROCEDURES FOR TYPESET OF SINGULAROBJECTS IN TEX
paramet.lib PROCEDURES FOR PARAMETRIZATION OF PRIMARY DECOMPOSITION
```

D.3 general_lib

```
LIBRARY:  general.lib   PROCEDURES OF GENERAL TYPE

A_Z("a",n);             string a,b,... of n comma seperated letters
binomial(n,m[,../..]); n choose m (type int), [type string/type number]
factorial(n[,../..]);  n factorial (=n!) (type int), [type string/number]
fibonacci(n[,p]);       nth Fibonacci number [char p]
kmemory();              int = active memory (kilobyte)
killall();              kill all user-defined variables
number_e(n);            compute exp(1) up to n decimal digits
number_pi(n);           compute pi (area of unit circle) up to n digits
primes(n,m);            intvec of primes p, n<=p<=m
product(../..[,v]);     multiply components of vector/ideal/...[indices v]
ringweights(r);         intvec of weights of ring variables of ring r
sort(ideal/module);     sort generators according to monomial ordering
sum(vector/id/..[,v]); add components of vector/ideal/...[with indices v]
which(command);         searches for command and returns absolute
                       path, if found
         (parameters in square brackets [] are optional)
```

D.4 matrix_lib

```
LIBRARY:  matrix.lib    PROCEDURES FOR MATRIX OPERATIONS

compress(A);            matrix, zero columns from A deleted
concat(A1,A2,..);       matrix, concatenation of matrices A1,A2,...
diag(p,n);              matrix, nxn diagonal matrix with entries poly p
dsum(A1,A2,..);         matrix, direct sum of matrices A1,A2,...
flatten(A);             ideal, generated by entries of matrix A
genericmat(n,m[,id]);  generic nxm matrix [entries from id]
is_complex(c);          1 if list c is a complex, 0 if not
outer(A,B);             matrix, outer product of matrices A and B
power(A,n);             matrix/intmat, n-th power of matrix/intmat A
skewmat(n[,id]);        generic skew-symmetric nxn matrix [entries from id]
submat(A,r,c);          submatrix of A with rows/cols specified by intvec r/c
symmat(n[,id]);         generic symmetric nxn matrix [entries from id]
tensor(A,B);            matrix, tensor product of matrices A nd B
unitmat(n);             unit square matrix of size n
gauss_col(A);           transform constant matrix A into col-reduced nf
gauss_row(A);           transform constant matrix A into row-reduced nf
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
         (parameters in square brackets [] are optional)
```

D.5 sing_lib

```
LIBRARY:  sing.lib      PROCEDURES FOR SINGULARITIES

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
slocus(i);             ideal of singular locus of ideal i
spectrum(f,w);         spectrum numbers of w-homogeneous polynomial f
Tjurina(i);            SB of Tjurina module of ideal i (assume i is ICIS)
tjurina(i);            Tjurina number of ideal i (assume i is ICIS)
T1(i);                 T1-module of ideal i
T2((i);                T2-module of ideal i
T12(i);                T1- and T2-module of ideal i
```

D.6 elim_lib

```
LIBRARY:  elim.lib      PROCEDURES FOR ELIMINATIOM, SATURATION AND BLOWING UP

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 [..mth] variable
sat(id,j);             saturated quotient of ideal/module id by ideal j
select(id,n[,m]);      select generators containing nth [..mth] variable
         (parameters in square brackets [] are optional)
```

D.7 inout_lib

```
LIBRARY:  inout.lib     PROCEDURES FOR MANIPULATING IN- AND OUTPUT

allprint(list);        print list if ALLprint is defined, with pause if >0
dbpri(n,list);         print objects of list if int n<=printlevel
lprint(poly/...[,n]);  display poly/... fitting to pagewidth [size n]
pmat(matrix[,n]);      print form-matrix [first n chars of each colum]
rMacaulay(string);     read Macaulay_1 output and return its Singular format
show(any);             display any object in a compact format
showrecursive(id,p);   display id recursively with respect to variables in p
split(string,n);       split given string into lines of length n
tab(n);                string of n space tabs
```

```
writelist(fil,nam,L);  write the list L into a file ‘fil‘ and call it ‘nam‘
         (parameters in square brackets [] are optional)
```

D.8 random_lib

```
LIBRARY:  random.lib    PROCEDURES OF RANDOM MATRIX AND POLY OPERATIONS

genericid(id[,p,b]);   generic sparse linear combinations of generators of id
randomid(id,[k,b]);    random linear combinations of generators of id
randommat(n,m[,id,b]); nxm matrix of random linear combinations of id
sparseid(k,u[,o,p,b]); ideal of k random sparse poly's of degree d [u<=d<=o]
sparsemat(n,m[,p,b]);  nxm sparse integer matrix with random coefficients
sparsepoly(u[,o,p,b]); random sparse polynomial with terms of degree in [u,o]
sparsetriag(n,m[..]);  nxm sparse lower-triag intmat with random coefficients
         (parameters in square brackets [] are optional)
```

D.9 deform_lib

```
LIBRARY:  deform.lib       PROCEDURES FOR COMPUTING MINIVERSAL DEFORMATION
                          by Bernd Martin (martin@math.tu-cottbus.de)

versal(Fo[,d,any])        miniversal deformation of isolated singularity Fo
mod_versal(Mo,I,[,d,any]) miniversal deformation of module Mo modulo ideal I
lift_kbase(N,M);          lifting N into standard kbase of M
lift_rel_kb(N,M[,kbM,p])  relative lifting N into a kbase of M
kill_rings(["prefix"])    kills the exported rings from above

SUB-PROCEDURES            used by main procedure:
                get_rings,compute_ext,get_inf_def,interact1,
                interact2,negative_part,homog_test
```

D.10 homolog_lib

```
LIBRARY:  homolog.lib   PROCEDURES FOR HOMOLOGICAL ALGEBRA

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

D.11 poly_lib

```
LIBRARY:  poly.lib      PROCEDURES FOR MANIPULATING POLYS, IDEALS, MODULES

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 occuring 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
        (parameters in square brackets [] are optional)
```

D.12 ring_lib

```
LIBRARY:  ring.lib      PROCEDURES FOR MANIPULATING RINGS AND MAPS

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
ringtensor("R",s,t,..);create ring R, tensor product of rings s,t,...
         (parameters in square brackets [] are optional)
```

D.13 finvar_lib

```
LIBRARY:  finvar.lib       LIBRARY TO CALCULATE INVARIANT RINGS & MORE
                          (c) Agnes Eileen Heydtmann,
                          send bugs and comments to agnes@math.uni-sb.de

cyclotomic(...)                        cyclotomic polynomial
group_reynolds(...)                    finite group and Reynolds operator
```

```
molien(...)                              Molien series
reynolds_molien(...)                     Reynolds operator and Molien series
partial_molien(...)                      partial expansion of Molien series
evaluate_reynolds(...)                   image under the Reynolds operator
invariant_basis(...)                     basis of homogeneous invariants
invariant_basis_reynolds(...)            basis of homogeneous invariants
primary_char0(...)                       primary invariants
primary_charp(...)                       primary invariants
primary_char0_no_molien(...)             primary invariants
primary_charp_no_molien(...)             primary invariants
primary_charp_without(...)               primary invariants
primary_invariants(...)                  primary invariants
primary_char0_random(...)                primary invariants
primary_charp_random(...)                primary invariants
primary_char0_no_molien_random(...)      primary invariants
primary_charp_no_molien_random(...)      primary invariants
primary_charp_without_random(...)        primary invariants
primary_invariants_random(...)           primary invariants
power_products(...)                      exponents for power products
secondary_char0(...)                     secondary invariants
secondary_charp(...)                     secondary invariants
secondary_no_molien(...)                 secondary invariants
secondary_with_irreducible_ones_no_molien(...)
                                         secondary invariants
secondary_not_cohen_macaulay(...)        secondary invariants
invariant_ring(...)                      primary and secondary invariants
invariant_ring_random(...)               primary and secondary invariants
algebra_containment(...)                 answers query of algebra containment
module_containment(...)                  answers query of module containment
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
```

D.14 primdec_lib

```
LIBRARY: primdec.lib: PROCEDURE FOR PRIMARY DECOMPOSITION

minAssGTZ(I);     computes the minimal associated primes
                    via Gianni,Trager,Zacharias
minAssChar(I);    computes the minimal associated primes
                    using characteristic sets
primdecGTZ(I);    computes a complete primary decomposition
                    via Gianni,Trager,Zacharias
primdecSY(I);     computes a complete primary decomposition
                    via Shimoyama-Yokoyama
testPrimary(L,k); tests whether the result of the primary
                    decomposition is correct
radical(I);       computes the radical of the ideal I
equiRadical(I);   computes the radical of the equidimensional part
```

```
                    of the ideal I
prepareAss(I);    computes the radicals of the equidimensional parts of I
```

D.15 invar_lib

```
LIBRARY: invar.lib PROCEDURES FOR COMPUTING INVARIANTS OF (C,+)-ACTIONS

invariantRing(matrix m,poly p,poly q,int choose)
// ring of invariants of the action of the additive group
// defined by the vectorfield corresponding to the matrix m
// (m[i,1] are the coefficients of d/dx(i))
// the polys p and q are assumed to be variables x(i) and x(j)
// such that m[j,1]=0 and m[i,1]=x(j)
// if choose=0 the computation stops if generators of the ring
// of invariants are computed (to be used only if you know that
// the ring of invariants is finitey generated)
// if choose<>0 it computes invariants up to degree choose

actionIsProper(matrix m)
// returns 1 if the action of the additive group defined by the
// matrix m as above i proper and 0 if not.
```

D.16 latex_lib

```
LIBRARY: latex.lib    PROCEDURES FOR TYPESET OF SINGULAROBJECTS IN LATEX2E
                      by Christian Gorzel, send bugs and
                      comments to gorzelc@math.uni-muenster.de

closetex(fnm);       writes closing line for TeX-document
opentex(fnm);        writes header for TeX-file fnm
tex(fnm);            calls LaTeX2e for TeX-file fnm
texdemo([n]);        produces a file explaining the features of this lib
texfactorize(fnm,f); creates string in TeX-Symbolformat for factors of poly f
texmap(fnm,m,r1,r2); creates string in TeX-Symbolformat for map m:r1->r2
texname(fnm,s);      creates string in TeX-Symbolformat for identifier
texobj(l);           creates string in TeX-Symbolformat for any (basic) type
texpoly(f,n[,l]);    creates string in TeX-Symbolformat for poly
texproc(fnm,p);      creates string in TeX-Symbolformat of text from proc p
texring(fnm,r[,l]);  creates string in TeX-Symbolformat for ring/qring
rmx(s);              removes .aux and .log files of TeXfile s
xdvi(s);             calls xdvi for dvi-file s
       (parameters in square brackets [] are optional)

                    Global Variables:
TeXwidth, TeXnofrac, TeXbrack, TeXproj, TeXaligned, TeXreplace, NoDollars
                are used to control the typesetting
```

```
  Call example texdemo; to become familiar with the features of latex.lib


TeXwidth      : int: -1,0,1..9, >9  controls the breaking of long polynomials
TeXnofrac     : (int) flag,  write 1/2 instead of \frac{1}{2}
TeXbrack      : string: possible values {,(,<,|, the empty string
                        controls brackets around ideals and matrices
TeXproj       : (int) flag, write : instead of , in intvecs and vectors
TeXaligned    : (int) flag, write maps (and ideals) aligned
TeXreplace    : list, entries twoelemented list for replacing symbols
NoDollars     : (int) flag, suppresses surrounding $ signs
```

D.17 hnoether_lib

```
LIBRARY:  hnoether.lib   PROCEDURES FOR THE HAMBURGER-NOETHER-DEVELOPMENT

         Important procedures:
develop(f [,n]);    Hamburger-Noether development from the irred. polynomial f
reddevelop(f);      Hamburger-Noether development from the (red.) polynomial f
extdevelop(hne,n);  extension of Hamburger-Noether development hne from f
param(hne);         returns a parametrization of f (input=output(develop))
displayHNE(hne);    display Hamburger-Noether development as an ideal
invariants(hne);    invariants of f, e.g. the characteristic exponents
displayInvariants(hne);  display invariants of f
generators(hne);    computes the generators of the semigroup of values
intersection(hne1,hne2); intersection multiplicity of two curves
stripHNE(hne);      reduce amount of memory consumed by hne
```

D.18 classify_lib

```
LIBRARY:  classify.lib  PROCEDURES FOR THE ARNOLD-CLASSIFIER OF SINGULARITIES

 A library for classifying isolated hypersurface singularities w.r.t. right
 equivalence, based on the determinator of singularities by V.I. Arnold.
 Author: Kai Krueger, krueger@mathematik.uni-kl.de
 last modified: 04.04.1998

basicinvariants(f);  computes Milnor number, determinacy-bound and corank 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 acoording to the number repetitions
init_debug([n]);     print trace and debugging information depending on int n
internalfunctions(); display names of internal procedures of this library
milnorcode(f[,e]);   Hilbert 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
tschirnhaus(f,v);    Tschirnhaus transformation of f w.r.t. variable v
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)
```

D.19 graphics_lib

```
LIBRARY: graphics.lib    PROCEDURES FOR GRAPHICS WITH MATHEMATICA

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

D.20 normal_lib

```
LIBRARY: normal.lib: PROCEDURE FOR NORMALIZATION (I)

normal(ideal I)
// computes a set of rings such that their product is the
// normalization of the reduced basering/I
```

Appendix E Library function index

T

U

V

W

X

7 Index

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

Short Contents

Table of Contents