Singular

A Computer Algebra System for Polynomial Computations

Manual

Version 2-0-3

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

with contributions by
O. Bachmann, W. Decker, S. Endraß, I. Farran, A. Frühbis-Krüger, C. Gorzel,
H. Grassmann, A. Heydtmann, D. Hillebrand, T. Keilen, U. Klein, K. Krüger, M. Lamm,
C. Lossen, B. Martin, M. Meßollen, W. Neumann, T. Nüßler, W. Pohl, T. Siebert,
J. Schmidt, M. Schulze, R. Stobbe, M. Wenk, T. Wichmann

Fachbereich Mathematik
Zentrum für Computeralgebra
Universität Kaiserslautern
D-67653 Kaiserslautern

Short Contents

1 Preface

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

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

Please register yourself as a Singular user by using the registration form on the Singular homepage `http://www.singular.uni-kl.de`. If for some reason you cannot access the registration form, you can also register by sending an email to `singular@mathematik.uni-kl.de` with subject line `register` and body containing the following data: your name, email address, organisation, country and platform(s).

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

Availability

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

GMP, libreadline
: © Free Software Foundation
`ftp://ftp.gnu.ai.mit.edu` or its mirrors

MP
: © Gray/Kajler/Wang, Kent State University
`http://www.symbolicnet.org/areas/protocols/mp.html`

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

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

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

Cygwin `http://sourceware.cygnus.com/cygwin/`

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

Acknowledgements

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

Currently, the SINGULAR team has the following members: Olaf Bachmann, Anne Frübis-Krüger, Thomas Keilen, Kai Krüger, Christoph Lossen, Viktor Levandovskyy, Wilfred Pohl, Mathias Schulze, Eric Westenberger, and Tim Wichmann.

Past members of the SINGULAR team are: Hubert Grassmann, Wolfgang Neumann, Jens Schmidt, Rüdiger Stobbe.

Further contributions to SINGULAR were made by: Thomas Bayer, Isabelle Bermejo, Stephan Endraß, Jose Ignacio Farran Martin, Wolfram Decker, Philippe Gimenez, Christian Gorzel, Agnes Heydtmann, Dietmar Hillebrand, Tobias Hirsch, Martin Lamm, Bernd Martin, Michael Meßollen, Thomas Nüßler, Moritz Wenk.

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 well ordering) and Mora's algorithm (if the ordering is a tangent cone ordering) as special cases. Furthermore, it provides polynomial factorizations, resultant, characteristic set and gcd computations, syzygy and free-resolution computations, and many more related functionalities.

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

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

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

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

For the highlights of the new SINGULAR version 2-0-3 see Section 7.1 [News and changes], page 600.

2.2 How to use this manual

For the impatient user

In Section 2.3 [Getting started], page 5, some simple examples explain how to use SINGULAR in a step-by-step manner.

Appendix A [Examples], page 231 should come next for real learning-by-doing or to quickly solve some given mathematical problems without dwelling too deeply into SINGULAR. This chapter contains a lot of real-life examples and detailed instructions and explanations on how to solve mathematical problems using SINGULAR.

For the systematic user

In Chapter 3 [General concepts], page 14, all basic concepts which are important to use and to understand SINGULAR are developed. But even for users preferring the systematic approach it will be helpful to 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 14, and its subsections there are some words on entering and exiting SINGULAR, followed by a number of other aspects concerning the interactive user-interface.
- To do anything more than trivial integer computations, one needs to define a basering in SINGULAR. This is explained in detail in Section 3.3 [Rings and orderings], page 28.
- An overview of the algorithms implemented in the kernel of SINGULAR is given in Section 3.4 [Implemented algorithms], page 34.
- In Section 3.5 [The SINGULAR language], page 37, language specific concepts are introduced such as the notions of names and objects, data types and conversion between them, etc.
- In Section 3.6 [Input and output], page 44, 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 3.7 [Procedures], page 47, Section 3.8 [Libraries], page 51, and Section 3.10 [Debugging tools], page 63.

Chapter 4 [Data types], page 67, 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 121, 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 225, is a loose collection of limitations and features which may be unexpected by those who expect the SINGULAR language to be an exact copy of the C programming language or of some Computer Algebra system's languages. But some mathematical hints are collected there, as well.

Appendix C [Mathematical background], page 281, 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 293, lists the libraries which come with SINGULAR and the functions contained in them, respectively.

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, most of the 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 78, 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 [2,3] is: 6 .
s="m[2,1]=0; m;";
execute(s);
↦ 1,0,3,
↦ 0,5,6,
↦ 7,8,9
```

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

2.3.2 Rings and standard bases

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

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

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

Other ring declarations:

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

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

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

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

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

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

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

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

```
ring r8;
r8;
↦ //   characteristic : 32003
↦ //   number of vars : 3
↦ //        block   1 : ordering dp
↦ //                  : names    x y z
↦ //        block   2 : ordering C
```

Defining a ring makes this ring the current active basering, so each ring definition above switches to a new basering. The concept of rings in SINGULAR is discussed in detail in Section 3.3 [Rings and orderings], page 28.

The basering is now `r8`. Since we want to calculate in the ring `r`, which we defined first, we have to switch back to it. This can be done using the function `setring`:

```
setring r;
```

Once a ring is active, we can define polynomials. A monomial, say x^3 may be entered in two ways: either using the power operator `^`, saying `x^3`, or in short-hand notation without operator, saying `x3`. Note that the short-hand notation is forbidden if 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 lie on the variety `V(f,g)`.

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

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

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

The function `groebner`, like many other functions in SINGULAR, prints a protocol during calculations, if desired. The command `option(prot);` enables protocolling whereas `option(noprot);` turns it off. Section 5.1.89 [option], page 177, explains the meaning of the different symbols printed during calculations.

The command `kbase` calculates a basis of the polynomial ring modulo an ideal, if the quotient ring is finite dimensional. As an example we calculate the Milnor number of a hypersurface singularity in the global and local case. This is the vector space dimension of the polynomial ring modulo the Jacobian ideal in the global case resp. of the power series ring modulo the Jacobian ideal in the local case. See Section A.3 [Critical points], page 234, for a detailed explanation.

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

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

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

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

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

```
↦ K[11]=x
↦ K[12]=1
```

Then

```
size(K);
↦ 12
```

gives the desired vector space dimension $K[x,y,z]/\mathrm{jacob}(f)$. 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 274, and Appendix C [Mathematical background], page 281).

```
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 94.

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

```
Milnor(f);
↦ 4
```

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

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

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

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

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

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

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

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

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

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

2.3.5 Modules and their annihilator

Now we shall give three more advanced examples.

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

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

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

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

However the submodule MD may also be considered as the module of relations of the factor module r^3/MD. In this way, SINGULAR can treat arbitrary finitely generated modules over the basering (see Section B.1 [Representation of mathematical objects], page 274).

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

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

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

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

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

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

The result is too big to be shown here.

2.3.6 Resolution

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

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

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

Disable this protocol with

```
option(noprot);
```

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

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

```
↦ total:     1     4     5     2
```

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

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

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

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

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

3 General concepts

3.1 Interactive use

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

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

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

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

Generally, we recommend to use SINGULAR in its Emacs interface, since this offers many more features and is more convenient to use than the ASCII-terminal interface (see Section 3.2 [Emacs user interface], page 21).

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

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

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

See Section 3.1.6 [Command line options], page 18, Section 3.1.7 [Startup sequence], page 21, Section 3.2.2 [Running SINGULAR under Emacs], page 24.

3.1.2 The SINGULAR prompt

The SINGULAR prompt `>` (larger than) asks the user for input of commands. The "continuation" prompt `.` (period) asks the user for input of missing parts of a command (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 (i.e., the title page of the SINGULAR manual) which offers a short table of contents. Typing `help` topic`;` shows the available documentation on topic. Here, topic may be either a function name or, more generally, any index entry of the SINGULAR manual. Furthermore, topic may contain wildcard characters. See Section 5.1.43 [help], page 148, for more information.

Online help information can be displayed in various help browsers. The following table lists a summary of the supported browsers. See below for more details about particular browsers.

Browser	**Platform**	**Description**
html	Windows	displays html version of manual in your default html browser
winhlp	Windows	displays Windows help (.hlp) version of manual
netscape	Unix, Windows	uses `netscape` to display html version of manual
tkinfo	Unix	uses the `tkinfo` program to display texinfo version of manual
xinfo	Unix, Windows	uses the `info` program to display texinfo version of manual in an extra `xterm`
info	Unix, Windows	uses the `info` program to display texinfo version of manual
builtin	all	simply outputs the help information in plain ASCII format
emacs	Unix, Windows	when running SINGULAR within (X)emacs, display help in (X)emacs info buffer.
dummy	all	displays error message about no available help browser

The browser which is used to display the help information, can be either set at startup time with the command line option (see Section 3.1.6 [Command line options], page 18)

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

or with the SINGULAR command (see Section 5.1.124 [system], page 205)

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

The SINGULAR command

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

lists all available browsers and the command

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

returns the currently used browser.

If no browser is explicitly set by the user, then the first (w.r.t. the order of the browsers in the table above) available browser is chosen. In other words, the default browser is `netscape` for Unix platforms, and `html` for Windows, unless your local set-up does not allow to run it.

The `.singularrc` (see Section 3.1.7 [Startup sequence], page 21) file is a good place to set your default browser. Recall that if a file `$HOME/.singularrc` exists on your system, then the content of this file is executed before the first user input. Hence, putting

```
if (system("version") >= 1306) // keep backwards-compatibility
{
  if (! system("--emacs")
  {
    // only set help browser if not running within emacs
```

```
      system("--browser", "info");
    }
    // if help browser is later on set to netscape,
    // allow it to fetch HTML pages from the net
    system("--allow-net", 1);
  }; // the last semicolon is important: otherwise no ">", but "." prompt
```

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

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

- Under Unix, the environment variable `DISPLAY` has to be set for the browsers `netscape`, `tkinfo`, and `xinfo` to work.
- The help browsers `netscape`, `tkinfo`, `xinfo`, and `info` are only available if the respective programs are installed on your system (for `xinfo`, the programs `xterm` and `info` are necessary). You can explicitly specify which program to use, by setting appropriate environment variables (whose names are the names of the respective programs in all up-case letters).
  For example, to use the program `navigator` with a private colormap as `netscape` help browser, set the environment variable `NETSCAPE` to the value `"navigator -install"`.
- If the help browser `netscape` can not find the local html pages of the SINGULAR manual (which it looks for at `$RootDir/html` – see Section 3.8.1 [Loading of a library], page 51 for more info on `$RootDir`) *and* the (command-line) option `--allow-net` has *explicitly* been set (see Section 3.1.6 [Command line options], page 18 and Section 5.1.124 [system], page 205 for more info on setting values of command-line options), then it dispatches the html pages from `http://www.singular.uni-kl.de/Manual` (note that the non-local net-access of HTML pages is disabled, by default).
  An alternative location of a local directory where the html pages reside can be specified by setting the environment variable `SINGULAR_HTML_DIR`.
- The help browsers `tkinfo`, `xinfo`, `info`, and `builtin` need the (info) file `singular.hlp` which it looks for at `$RootDir/info/singular.hlp` (see Section 3.8.1 [Loading of a library], page 51 for more info on `$RootDir`). An alternative location of the info file of the manual can be specified by setting the environment variable `SINGULAR_INFO_FILE`.

See also: Section 3.1.6 [Command line options], page 18; Section 3.8.1 [Loading of a library], page 51; Section 3.1.7 [Startup sequence], page 21; Section 5.1.124 [system], page 205.

Info help browsers

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

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

To move within info, type commands consisting of single characters. Do not type `RETURN`. Do not use cursor keys, either. Using some of the cursor keys by accident might pop to some totally different node. Type `l` to return to the original node. Some of the `info` commands read input from the command line at the bottom 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 selects the node in which the next occurrence is found

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

3.1.4 Interrupting SINGULAR

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

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

`c` continues

`q` quits SINGULAR

3.1.5 Editing input

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

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

`CTRL-B` moves cursor 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 gives next line

`CTRL-P` saves current line on history and gives previous line

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

When run under a Unix-like operating system and in its ASCII-terminal user interface, Singular tries to dynamically link at runtime with the GNU Readline library. See section "Command Line Editing" in *The GNU Readline Library Manual*, for more information. If a shared version of this library can be found on your machine, then additional command-line editing features like history completion are available. In particular, if Singular is able to load that library and 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 ...]]
ESingular  [options] [file1 [file2 ...]]
```

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

`-d`, `--sdb` Enable the use of the source code debugger. See Section 3.10.2 [Source code debugger], page 64.

`-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 220.

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

`--allow-net`
Allow the `netscape` and `html` help browser to fetch HTML manual pages over the net from the WWW home-site of Singular. See Section 3.1.3 [The online help system], page 14, for more info.

`--browser=VAL`
Use `VAL` as browser for the Singular online manual.
`VAL` may be one of `html` (Windows only), `netscape`, `xinfo`, `tkinfo`, `info`, `builtin`, or `emacs`. Depending on your platform and local installation, only some browsers might be

available. The default browser is `html` for Windows and `netscape` for Unix platforms. See Section 3.1.3 [The online help system], page 14, for more info.

`--no-rc` Do not execute the `.singularrc` file on start-up. By default, this file is executed on start-up. See Section 3.1.7 [Startup sequence], page 21.

`--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 21.

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

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

`-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 enable the passing of commands and strings to SINGULAR:

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

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

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

`--min-time=SECS`
: If the `timer` (see Section 5.3.8 [timer], page 222), resp. `rtimer` (see Section 5.3.10 [rtimer], page 224) , 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 next 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 88.

`--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 88.

`--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 88.

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

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

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

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

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

The value of options given to SINGULAR (resp. their default values, if an option was not given), can be checked with the command `system("--long_option_name")`. See Section 5.1.124 [system], page 205.

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

Furthermore, the value of options (e.g., `--browser`) can be re-set while SINGULAR is running with the command `system("--long_option_name_string ",expression)`. See Section 5.1.124 [system], page 205.

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

3.1.7 Startup sequence

On start-up, Singular

1. loads the library `standard.lib` (provided the `--no-stdlib` option was not given),
2. searches the current directory and then the home directory of the user, and then all directories contained in the library `SearchPath` (see Section 3.8.1 [Loading of a library], page 51 for more info on `SearchPath`) for a file named `.singularrc` and executes it, if found (provided the `--no-rc` option was not given),
3. executes the string specified with the `--execute` command line option,
4. executes the files `file1`, `file2` ... (given on the command line) in that order.

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

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

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

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

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

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

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

3.2 Emacs user interface

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

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

- color-highlighting
- truncation of long lines
- folding of input and output

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

To use the SINGULAR-Emacs interface you need to have Emacs version 20 or higher, or XEmacs version 20.3 or higher installed on your system. These editors can be downloaded for most hard- and software platforms (including Windows 95/98/NT, but excluding the Macintosh), from either `http://www.fsf.org/software/emacs/emacs.html` (Emacs), from `http://www.xemacs.org` (XEmacs), or from our ftp site at `ftp://www.mathematik.uni-kl.de/pub/Math/Singular/utils/`. The differences between Emacs and XEmacs w.r.t. the SINGULAR-Emacs interface are marginal – which editor to use is mainly a matter of personal preferences.

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

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

3.2.1 A quick guide to Emacs

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

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

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

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

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

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

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

  `C-File->Un-Split` Un-Split window (i.e., kill other windows)
  `x`
  `1`

`C-x o`	Goto other window, i.e. move cursor into other window.

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

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

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

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

The most important buffer commands are

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

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

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

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

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

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

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

3.2.2 Running SINGULAR under Emacs

There are two ways to start the SINGULAR Emacs interface: Typing `ESingular` instead of `Singular` on the command shell launches a new Emacs process, initializes the interface and runs SINGULAR within Emacs. The other way is to start the interface in an already running Emacs, by typing `M-x singular` inside Emacs. This initializes the interface and runs SINGULAR within Emacs. Both ways are described in more detail below.

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

Starting the interface using ESingular

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

The customizable variables of the SINGULAR Emacs interface are set to defaults which give the novice user a very shell like feeling of the interface. Nevertheless, these default settings can be changed, see Section 3.2.4 [Customization of the Emacs interface], page 26. Besides other Emacs initializations, such as fontification or blinking parentheses, a new menu item called `Singular` is added to the main menu, providing menu items for starting SINGULAR. On XEmacs, a button starting SINGULAR is added to the main toolbar.

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

`ESingular` inherits all `Singular` options. For a description off all these options, see Section 3.1.6 [Command line options], page 18. Additionally there are the following options which are special to `ESingular`:

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

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

Starting the interface within a running Emacs

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

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

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

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

Starting, interrupting and stopping SINGULAR

There are the following commands to start and stop SINGULAR:

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

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

  1. The Singular executable. This can either be a file name with complete path, e.g., `/local/bin/Singular`. Then exactly this executable is started. The path may contain the character `~` denoting your home directory. Or it can be the name of a command without path, e.g., `Singular`. Then the executable is searched for in your `$PATH` environment variable.
  2. The default working directory. This is the path to an existing directory, e.g., `~/work`. The current directory is set to this directory before SINGULAR is started.
  3. Command line options. You can set any SINGULAR command line option (see Section 3.1.6 [Command line options], page 18).
  4. The buffer name. You can specify the name of the buffer the interface is running in.

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

  Starts SINGULAR with default settings for the executable, the working directory, command line switches, and the buffer name. You can customize this default settings, see Section 3.2.4 [Customization of the Emacs interface], page 26.

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

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

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

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

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

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

Whenever a SINGULAR process is started within the Emacs interface, the contents of a special startup file (by default `~/.emacs-singularrc`) is pasted as input to SINGULAR at the very end of the usual startup sequence (see Section 3.1.7 [Startup sequence], page 21). The name of the startup file can be changed, see Section 3.2.4 [Customization of the Emacs interface], page 26.

3.2.3 Demo mode

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

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

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

Some aspects of running SINGULAR demos can be customized. See Section 3.2.4 [Customization of the Emacs interface], page 26, for more info.

3.2.4 Customization of the Emacs interface

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

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

The SINGULAR interface customization buffer is divided into four groups:

- Singular Faces

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

- Singular Sections And Foldings

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

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

- Singular Interactive Miscellaneous

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

- Singular Demo Mode

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

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

3.2.5 Editing SINGULAR input files with Emacs

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

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

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

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

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

3.2.6 Top 20 Emacs commands

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

Starting and stopping of SINGULAR:

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

Editing input and output:

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

Loading of files and SINGULAR demo mode:

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

Help and Customization:

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

3.3 Rings and orderings

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

1. a polynomial ring over a field,
2. a 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 3.3.3 [Term orderings], page 32, Appendix C [Mathematical background], page 281.

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 floating point numbers of a user defined precision.
7. the field of complex numbers represented by (pairs of) floating point numbers of a user defined precision.

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 5.2.8 [keepring], page 217, Section 5.1.113 [setring], page 196.

Objects of ring dependent types are local to a ring. To access them after a change of the basering they have to be mapped using `map` or by the functions `imap` or `fetch`. See Section 3.5.4 [Objects], page 42, Section 5.1.31 [fetch], page 141, Section 5.1.48 [imap], page 152, Section 4.8 [map], page 94.

All changes of the basering in a procedure are local to this procedure unless a `keepring` command is used as the last statement of the procedure. See Section 3.7 [Procedures], page 47, Section 5.2.8 [keepring], page 217.

3.3.1 Examples of ring declarations

The exact syntax of a ring declaration is given in the next two subsections; this subsection lists some examples first. Note that the ordering has to be chosen such that the unit-elements of the ring are precisely those elements with leading monomial 1. For more information, see Section B.2 [Monomial orderings], page 275.

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

- the ring $Z/32003[x, y, z]$ with degree reverse lexicographical ordering. The exact ring declaration may be omitted in the first example since this is the default ring:
  ```
  ring 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,x_2,x_3 and degree reverse lexicographical ordering for x_4,x_5,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, respectively, and vectors are first ordered by components (in descending order) and then by monomials:

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

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

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

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

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

  refers to some generator of the cyclic group of units of K :

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

- the ring $R[x,y,z]$, where R denotes the field of real numbers represented by simple precision floating point numbers. This is a special case:

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

- the ring $R[x,y,z]$, where R denotes the field of real numbers represented by floating point numbers of 50 valid decimal digits and the same number of digits for the rest:

  ```
  ring r = (real,50),(x,y,z),dp;
  ```

- the ring $R[x,y,z]$, where R denotes the field of real numbers represented by floating point numbers of 10 valid decimal digits and with 50 digits for the rest:

  ```
  ring r = (real,10,50),(x,y,z),dp;
  ```

- the ring $R(j)[x,y,z]$, where R denotes the field of real numbers represented by floating point numbers of 30 valid decimal digits and the same number for the rest. j denotes the imaginary unit.

  ```
  ring r = (complex,30,j),(x,y,z),dp;
  ```

- the ring $R(i)[x,y,z]$, where R denotes the field of real numbers represented by floating point numbers of 6 valid decimal digits and the same number for the rest. i is the default for the imaginary unit.

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

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

  ```
  ring R = 7,(x,y,z), dp;
  qring r = std(maxideal(2));
  ```

3.3.2 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 current 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 220.
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. an expression_list of the name `real` and two optional int_expression giving the precision in decimal digits and the size for the stabilizing rest. The default for the rest is the same size as for the representation. An exception is the name `real` without any integers. These numbers are implemented as machine floating point numbers of single precision. Note that computations over all these fields are not exact.
5. an expression_list of the name `complex`, two optional int_expression and a name. This specifies the field of complex numbers represented with floating point numbers with a precision similar to `real`. An expression_list without int_expression defines a precision and rest with length 6. The name of the imaginary unit is given by the last parameter. Note that computations over these fields are not exact.

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

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

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

For the definition of the orderings, see Section B.2 [Monomial orderings], page 275.

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 5.1.31 [fetch], page 141).

SINGULAR computes in a quotient ring as long as possible with the given representative of a polynomial, say, `f`. I.e., it usually does not reduce `f` w.r.t. the quotient ideal. This is only done when necessary during standard bases computations or by an explicit reduction using the command `reduce(f, std(0))` (see Section 5.1.106 [reduce], page 191).

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
```

3.3.3 Term orderings

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

Remark: The novice user should generally use the ordering `dp` for computations in the polynomial ring $K[x_1,\ldots,x_n]$, resp. `ds` for computations in the localization $\mathrm{Loc}_{(x)}K[x_1,\ldots,x_n]$). For more details, see Appendix B [Polynomial data], page 274.

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 a `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) > ...`. It is not necessary to specify the module ordering explicitly since `(` ordering`, ..., C )` is the default.

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

For a mathematical description of these orderings, see Appendix B [Polynomial data], page 274.

3.4 Implemented algorithms

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

Nonetheless, there are a lot of other important algorithms:

- Algorithms to compute the standard operations on ideals and modules: intersection, ideal quotient, elimination, etc.
- Different Syzygy algorithms and algorithms to compute free resolutions of modules.
- Combinatorial algorithms to compute dimensions, Hilbert series, multiplicities, etc.
- Algorithms for univariate and multivariate polynomial factorization, resultant and gcd computations.

Commands to compute standard bases

`facstd`
: Section 5.1.29 [facstd], page 139
computes a list of Groebner bases via the Factorizing Groebner Basis Algorithm, i.e., their intersection has the same radical as the original ideal. It need not be a Groebner basis of the given ideal.

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

`fglm`
: Section 5.1.32 [fglm], page 142
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 5.1.42 [groebner], page 147
computes a standard resp. Groebner bases using a heuristically chosen method.

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

`mstd`
: Section 5.1.79 [mstd], page 171
computes a standard basis and a minimal set of generators.

`std`
: Section 5.1.120 [std], page 202
computes a standard resp. Groebner basis.

`stdfglm`
: Section 5.1.121 [stdfglm], page 204
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 5.1.122 [stdhilb], page 204
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 132
computes the (Krull) dimension, codimension and the multiplicity.

The result is only displayed on the screen.

`dim` Section 5.1.19 [dim], page 134
computes the dimension of the ideal resp. module.

`highcorner`
Section 5.1.44 [highcorner], page 149
computes the smallest monomial not contained in the ideal resp. module. The ideal resp. module has to be finite dimensional as a vector space over the ground field.

`hilb` Section 5.1.45 [hilb], page 149
computes the first, and resp. or, second Hilbert series of an ideal resp. module.

`kbase` Section 5.1.56 [kbase], page 157
computes a vector space basis (consisting of monomials) of the quotient of a ring by an ideal resp. of a free module by a submodule.

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

`mult` Section 5.1.80 [mult], page 172
computes the degree of the monomial ideal resp. module generated by the leading monomials of the input.

`reduce` Section 5.1.106 [reduce], page 191
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 5.1.134 [vdim], page 212
computes the vector space dimension of a ring (resp. free module) modulo an ideal (resp. module).

Commands to compute resolutions

`res` Section 5.1.109 [res], page 193
computes a free resolution of an ideal or module using a heuristically chosen method. This is the preferred method to compute free resolutions of ideals or modules.

`lres` Section 5.1.69 [lres], page 165
computes a free resolution of an ideal or module with La Scala's method. The input needs to be homogeneous.

`mres` Section 5.1.78 [mres], page 171
computes a minimal free resolution of an ideal or module with the Syzygy method.

`sres` Section 5.1.118 [sres], page 200
computes a free resolution of an ideal or module with Schreyer's method. The input has to be a standard basis.

`nres` Section 5.1.85 [nres], page 175
computes a free resolution of an ideal or module with the standard basis method.

`syz` Section 5.1.125 [syz], page 207
computes the first Syzygy (i.e., the module of relations of the given generators).

Further processing of resolutions

`betti` Section 5.1.3 [betti], page 124
computes the graded Betti numbers of a module from a free resolution.

`minres` Section 5.1.74 [minres], page 168
minimizes a free resolution of an ideal or module.

`regularity`
Section 5.1.107 [regularity], page 192
computes the regularity of a homogeneous ideal resp. module from a given minimal free resolution.

Processing of polynomials

`char_series`
Section 5.1.5 [char_series], page 126
computes characteristic sets of polynomial ideals.

`extgcd` Section 5.1.28 [extgcd], page 139
computes the extended gcd of two polynomials.

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

`factorize`
Section 5.1.30 [factorize], page 140
computes factorization of univariate and multivariate polynomials into irreducible factors.

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

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

`gcd` Section 5.1.39 [gcd], page 146
computes greatest common divisors of univariate and multivariate polynomials.

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 for polynomials over algebraic extensions of Q.

`resultant`
Section 5.1.111 [resultant], page 195
computes the resultant of two univariate polynomials using the subresultant algorithm.

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

`vandermonde`
Section 5.1.131 [vandermonde], page 210
interpolates a polynomial from its values at several points

Matrix computations

`bareiss` Section 5.1.2 [bareiss], page 122
implements sparse Gauss-Bareiss method for elimination (matrix triangularization) in arbitrary integral domains.

`det` Section 5.1.17 [det], page 133
computes the determinant of a square matrix.

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

`minor` Section 5.1.73 [minor], page 168
computes all minors (=subdeterminants) of a given size for a matrix.

Numeric computations

`laguerre` Section 5.1.60 [laguerre], page 159
computes all (complex) roots of a univariate polynomial

`uressolve`
Section 5.1.130 [uressolve], page 210
find all roots of 0-dimensional ideal i with multivariate resultants

Controlling computations

`option` Section 5.1.89 [option], page 177
allows setting of options for manipulating the behavior of computations (such as reduction strategies) and for showing protocol information indicating the progress of a computation.

3.5 The SINGULAR language

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

In the following, the basic language concepts such as commands, expressions, names, objects, etc., are discussed. See Section 3.7 [Procedures], page 47, and Section 3.8 [Libraries], page 51, 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 225.

Elements of the language

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

- An "expression" is a sequence of operators, functions, and operands that specifies a computation. An expression always results in a value of a specific type. See Chapter 4 [Data types], page 67, and its subsections (e.g., Section 4.12.2 [poly expressions], page 107), for information on how to build expressions.
- A "command" is either a declaration, an assignment, a call to a function without return value, or a print command. For detailed information, see Section 3.5.1 [General command syntax], page 38.
- "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 214, 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.

3.5.1 General command syntax

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

Declaration

1. type name = expression ;
   declares a variable with the given name of the given type and assigns the expression as initial value to it. Expression is an expression of the specified type or one that can be converted to that type. See Section 3.5.5 [Type conversion and casting], page 42.
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 3.5.5 [Type conversion and casting], page 42.
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 3.5.3 [Names], page 41, for more information on declarations. See Chapter 4 [Data types], page 67, for a description of all data types known to SINGULAR.

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

Assignment

4. name = expression ;
   assigns expression to name.

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

There must be a type conversion of the type of expression to the type of name. See Section 3.5.5 [Type conversion and casting], page 42.

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

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

The function may have output (not to be confused with a return value of type string). See Section 5.1 [Functions], page 121. 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
↦ 2
```

Print command

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

Use the function `print` (or the procedure show from inout.lib) to get a pretty output of various data types, e.g., matrix or intmat. See Section 5.1.97 [print], page 183.

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

3.5.2 Special characters

The following characters and operators have special 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 228, 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 5.1.34 [filecmd], page 143.

`!` 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.

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

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

`;` statement separator

`,` separator for expression lists and function arguments

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

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

`_` value of expression last displayed

`~` breakpoint in procedures

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

`$` terminates SINGULAR

3.5.3 Names

Singular is a strongly typed language. This means that all names (= identifiers) have to be declared prior to their use. For the general syntax of a declaration, see the description of declaration commands (see Section 3.5.1 [General command syntax], page 38).

See Chapter 4 [Data types], page 67, for a description of Singular's data types. See Section 5.1.129 [typeof], page 209, 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 5.1.128 [type], page 209).

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 to this rule, the characters `@`, and `_` 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 5.1.110 [reservedName], page 194. Names should not interfere with names of ring variables or, more generally, with monomials. See Section 6.4 [Identifier resolution], page 229.
The command `listvar` provides a list of the names in use (see Section 5.1.68 [listvar], page 164).

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 referred to as name substitution.

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

3.5.4 Objects

Every object in SINGULAR has a type and a value. In most cases it has also a name and in some cases an attribute list. The value of an object may be examined simply by printing it with a print command: object;. The type of an object may be determined by means of the `typeof` function, the attributes by means of the `attrib` function (Section 5.1.129 [typeof], page 209, Section 5.1.1 [attrib], page 121):

```
  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 94, Section 5.1.31 [fetch], page 141, Section 5.1.48 [imap], page 152.

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

3.5.5 Type conversion and casting

Type conversion

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

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

9.	`int`	↦ `matrix`
10.	`intmat`	↦ `matrix`
11.	`intvec`	↦ `matrix`
12.	`module`	↦ `matrix`
13.	`number`	↦ `matrix`
14.	`poly`	↦ `matrix`
15.	`vector`	↦ `matrix`
16.	`ideal`	↦ `module`
17.	`matrix`	↦ `module`
18.	`vector`	↦ `module`
19.	`int`	↦ `number`
20.	`int`	↦ `poly`
21.	`number`	↦ `poly`
22.	`string`	↦ `proc`
23.	`list`	↦ `resolution`
24.	`int`	↦ `vector` (i ↦ i*`gen(1)`)
25.	`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:

to	from
`ideal`	expression lists of `int`, `number`, `poly`
`ideal`	`int`, `matrix`, `module`, `number`, `poly`, `vector`
`int`	`number`, `poly`
`intvec`	expression lists of `int`, `intmat`
`intmat`	`intvec` (see Section 4.4.3 [intmat type cast], page 79)
`list`	expression lists of any type
`matrix`	`module`, `ideal`, `vector`, `matrix`. There are two forms to convert something to a matrix: if `matrix(` expression `)` is used then the size of the matrix is determined by the size of expression. But `matrix(` expression `,` m `,` n `)` may also be used - the result is a $m \times n$ matrix (see Section 4.9.3 [matrix type cast], page 98)
`module`	expression lists of `int`, `number`, `poly`, `vector`
`module`	`ideal`, `matrix`, `vector`
`number`	`poly`
`poly`	`int`, `number`
`string`	any type (see Section 4.17.3 [string type cast], page 116)

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]
```

3.5.6 Flow control

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

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

Blocks are used whenever SINGULAR is used as a structured programming language. The `if` and `else` structures allow conditional execution of blocks (see Section 5.2.7 [if], page 217, Section 5.2.4 [else], page 215). `for` and `while` loops are available for repeated execution of blocks (see Section 5.2.6 [for], page 216, Section 5.2.11 [while], page 219). In procedure definitions the main part and the example section are blocks as well(see Section 4.13 [proc], page 110).

3.6 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.

For loading of libraries, see Section 5.1.65 [LIB], page 162. For executing program scripts, see Section 5.1.34 [filecmd], page 143.

Monitoring

A special form of I/O is monitoring. When monitoring is enabled, SINGULAR makes a typescript of everything printed on your terminal to a file. This is useful to create a protocol of a SINGULAR session. The `monitor` command enables and disables this feature (see Section 5.1.76 [monitor], page 170).

How to use links

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

For more information, see Section 5.1.137 [write], page 213, Section 5.1.105 [read], page 190, Section 5.1.21 [dump], page 135, Section 5.1.41 [getdump], page 146.

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 5.1.88 [open], page 177, Section 5.1.8 [close], page 127).

Links have various properties which can be queried using the `status` function (see Section 5.1.119 [status], page 201).

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 that can be written into files for storage or communication with other programs. The data are written in plain ASCII format. Reading from an ASCII link returns a string — conversion into other data is up to the user. This can be done, for example, using the command `execute` (see Section 5.1.26 [execute], page 138).

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 85, 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 86, 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 computer or on different ones. 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.28 [Parallelization with MPtcp links], page 272).

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

See Section 4.6.5.2 [MPtcp links], page 88, 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, the value associated to it is returned. Both the key and the value have to be specified as strings. Hence, DBM links may be used only for data which may be converted to or from strings.

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

See Section 4.6.6 [DBM links], page 90, for more information.

Example:

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

3.7 Procedures

Procedures contain sequences of commands of the Singular language. They are used to extend the set of commands with user defined commands. Procedures are defined by either typing them on the command line or by loading them from a so-called library file with the LIB command, see Section 5.1.65 [LIB], page 162. 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);`.

3.7.1 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 3.7.3 [Parameter list], page 49. 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 LIB command (Section 5.1.65 [LIB], page 162).
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 acce\
```

```
   ssed by an user.
↦    ? error occurred in line 5: `  "*"+internal_tab(3)+"*";`
  // show the help section for tab
  help tab;
↦ // ** Could not get IdxFile.
↦ // ** Either set environment variable SINGULAR_IDX_FILE to IdxFile,
↦ // ** or make sure that IdxFile is at /home/hannes/singular/200/doc/singu\
   lar.idx
↦ // 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 text of a procedure

There are no enforced rules on the format of the help section of a procedure.

Nevertheless, we recommend that 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 help text of procedures contained in libraries of the SINGULAR distribution should furthermore comply with certain rules as explained in Section 3.9.5 [The help string of procedures], page 59.

3.7.2 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.

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 it 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 5.1.68 [listvar], page 164). A ring may be 'moved up' by one nesting level with `keepring` (see Section 5.2.8 [keepring], page 217). All variables living in that ring are moved together with that ring.

To keep local variables after leaving the procedure, they have to be exported (i.e. made known) to some higher level by the `export` command (see Section 5.2.5 [export], page 216).

Example:

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

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

3.7.3 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 at least 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
}
```

3.7.4 Procedure commands

Some commands only make sense inside a procedure, since they make objects known to the nesting level from which the procedure was called or to all nesting levels.

See Section 5.2.5 [export], page 216; Section 5.2.8 [keepring], page 217; Section 5.2.10 [return], page 218.

3.8 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 libname_lib`. 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 libraries have to comply with the few syntax rules described below.

Furthermore, libraries which are to be included in the SINGULAR distribution, have to comply with certain rules as explained in Section 3.9 [Guidelines for writing a library], page 53.

3.8.1 Loading of a library

Libraries can 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 `/` and can not be located in the current directory, each directory contained in the library `SearchPath` is searched for file of this name.

Note on SearchPath:
The the library `SearchPath` is constructed at SINGULAR start-up time as follows:

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

For setting environment variables see Section 5.1.124 [system], page 205, or consult the manual of your shell.

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

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

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 precisely, option `-q` (and likewise `--quiet`) unsets option `loadLib` to inhibit monitoring of library loading (see Section 5.1.89 [option], page 177).

All loaded libraries are displayed by the `LIB;` command:

```
  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 inout.lib (1.21.2.3,2002/02/20)
  LIB;
↦ standard.lib,inout.lib
```

See Section 3.1.6 [Command line options], page 18; Section 5.1.65 [LIB], page 162; Section 2.3.3 [Procedures and libraries], page 9; Appendix D [SINGULAR libraries], page 293; Section 4.13 [proc], page 110; Section D.1 [standard_lib], page 293; Section 4.17 [string], page 114; Section 5.1.124 [system], page 205.

3.8.2 Format of a library

A library file can contain comments, a category-, info- and version-string definition, `LIB` commands, `proc` commands and `proc` commands with `example` and `help` sections, i.e., the following keywords are allowed: `category`, `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 category-, 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 libname_lib;`.

Example:

```
info="
      This could be the general help of a library.
     Quotes must be escaped with a \ 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).

Although there is no enforced format of the info string of a library, we recommend that you follow certain rules as explained in Section 3.9.4 [The help string of a library], page 59.

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.

Syntax: `category` = string_constant ;

Purpose: defines the category for the library.

Example:

```
category="Utilities";
```

Note: reserved for sorting the libraries into categories.

3.9 Guidelines for writing a library

Although there are very few enforced rules on how libraries are written (see Section 3.8 [Libraries], page 51), it is recommended that the libraries comply with the guidelines explained in this section, so that debugging and understanding are made easier.

Note: For libraries which are to be included in the SINGULAR distribution, the following guidelines are mandatory.

3.9.1 Procedures in a library

In this section we list miscellaneous recommendations on how procedures contained in a library should be implemented.

1. The info- and version-string should appear at the beginning of the library, before procedure definitions.
2. The info-string should have the format as explained in Section 3.9.4 [The help string of a library], page 59.
3. Each procedure which should not be accessible by users should be declared `static`.
4. Each procedure which is not declared `static` should have a help and example section as explained in Section 3.7.1 [Procedure definition], page 47.
   Such procedures should furthermore carefully check any assumptions made about their input (like the type of list elements), and, if necessary, omit an error using the function Section 5.1.24 [ERROR], page 137.
5. No procedures should be defined within the body of another procedure.
6. If the value of the reserved variable `printlevel` (see Section 5.3.6 [printlevel], page 221) is greater than 0 then interactive user-input, i.e., the usage of functions like `pause("..")` or `read("");` (see Section 5.1.105 [read], page 190), may be requested.
7. If the value of the reserved variable `printlevel` (see Section 5.3.6 [printlevel], page 221) is 0 then interactive user-input, i.e., the usage of functions like `pause("..")` or `read("");` (see Section 5.1.105 [read], page 190), may **not** be requested. Instead, an error (using the function Section 5.1.24 [ERROR], page 137) should be reported together with the recommendation on increasing the value of the reserved variable `printlevel`.
8. It is often useful for a procedure to print out comments, either to explain results or to display intermediate computations. However, if this procedure is called by another procedure, such comments are confusing and disturbing in most cases.

   SINGULAR offers an elegant solution, which requires the usage of the SINGULAR function Section 5.1.12 [dbprint], page 130 and the reserved variables Section 5.3.6 [printlevel], page 221, and Section 5.3.11 [voice], page 224 (`voice` counts the nesting of procedures; It has the value 1 on the top level, 2 inside the first procedure etc.; `printlevel` has the value 0 by default, but can be set to any integer value by the user).

For example, if the following procedure `Test` is called directly from the top level then '`comment1`' is displayed (i.e., printed out) but not '`comment2`'; and nothing is displayed if `Test` is called from within any other procedure. However, if `printlevel` is set to a value k with k>0, then '`comment1`' (resp. '`comment2`') is displayed provided that `Test` is called from other procedures, with a nesting level up to k (resp. k-1).

Note furthermore, that the example part of a procedure behaves in this respect like a procedure (i.e., the value of `voice` is 1). Therefore, the command `printlevel=1;` is necessary for '`comment1`' to be displayed on `example Test;`. However, since printlevel is a global variable, it should be reset to the old value at the end of the example part.

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

3.9.2 Documentation of a library

The typesetting language in which the Singular documentation is written is `texinfo`. Based on various tools, `info`, `dvi`, `ps`, and `html` versions of the `texinfo` documentation are generated.

Starting with Singular version 1-3, the `texinfo` documentation of all libraries of the Singular distribution is generated automatically from their source code.

More precisely, for each library,

- the info string of the library is parsed and typeset as explained in Section 3.9.3 [Typesetting of help strings], page 56.
- the help string of each procedure listed in the `PROCEDURE:` section of the library info string is parsed and typeset as explained in Section 3.9.3 [Typesetting of help strings], page 56.
- the example of each procedure listed in the `PROCEDURE:` section of the library info string is computed and its output is included into the documentation.

For a uniform look-and-feel of the library documentation, library developers should

- follow the recommendation of Section 3.9.4 [The help string of a library], page 59 and Section 3.9.5 [The help string of procedures], page 59.

- consult the source code of libraries like `template.lib` (see Section 3.9.6 [template_lib], page 59) for examples on how library documentations are written.
- make sure that each procedure listed in the `PROCEDURE:` section of the library info string has a help string and an example section.
- not use interactive functions like `pause("..")` or `read("");` (see Section 5.1.105 [read], page 190) and should limit the length of input lines to 60 characters in the example section of procedures.
- carefully check the generated documentation of their libraries in its various formats using the `lib2doc` (see Section 3.9.2.1 [lib2doc], page 55) utility.

3.9.2.1 lib2doc

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

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

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

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

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

1. Unpack lib2doc.tar.gz

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

   and

   ```
   cd lib2doc
   ```

2. Edit the beginning of the file `Makefile`, filling in the values for `SINGULAR` and `LIBPARSE`. Check also the values of `PERL` and `LATEX2HTML`.
3. Copy your library to the current directory:

   ```
   cp <path-where-your-lib-is>/mylib.lib .
   ```

4. Now you can run the following commands:

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

     ```
     info -f mylib.hlp
     ```

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

     ```
     xdvi mylib.dvi
     ```

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

     ```
     ghostview mylib.dvi
     ```

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

```
netscape mylib.html
```

`make clean`
: Deletes all generated files.

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

3.9.3 Typesetting of help strings

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

For optimal typesetting results, the guidelines for writing libraries and procedures should be followed, and the following points should be kept in mind:

- If a help string starts with an `@` sign, then no parsing is done, and the help string is assumed to be already in the texinfo format.
- help strings are typeset within a `@table @asis` environment (which is similar to a latex `description` environment).
- If a line starts with only uppercase words and contains a colon, then the text up to the colon is taken to be the description-string of an item and the text following the colon is taken to be the content of the item.
- If the description-string of an item matches

  `EXAMPLE` then this item and its content is ignored.

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

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

  `PROCEDURES`
  then the content of the item is assumed to be a summary description of the procedures contained in the library. Separate texinfo nodes (subsections in printed documents) are **only** created out of the help strings of such procedures which appear in the summary description of a library.

  `LIBRARY` then the content of the item is assumed to be a one-line description of a library. If this one-line description consist of only uppercase characters, then it is typeset in all lowercase characters in the manual (otherwise it is left as is).
- For the content of an item, the following texinfo markup elements are recognized (and, their content not further manipulated):

  `@*` to enforce a line-break.

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

  `@ref{...}`
  References to other parts of the SINGULAR manual can be set using one of the following `@ref{node}` constructs. Notice that `node` must be the name of a section

of the SINGULAR manual. In particular, it may be a name of a function, library or library procedure.

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

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

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

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

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

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

`@math{..}`
: for typesetting of small (i.e., which do not go over multiple lines) mathematical expressions in LaTeX math-mode syntax.

Example: `@math{\alpha}`
↦
α

Note: Mathematical expressions inside `@math{..}` may not contain curly parenthesis and the "at" sign, i.e., may not contain `{`,`}`,`@`.

`@code{..}`
: for typesetting of small (i.e., which do not go over multiple lines) strings in typewriter font.

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

Note: The string inside `@code{..}` may not contain curly parenthesis and the "at" sign, i.e., may not contain `{`,`}`,`@`.

`@example` ...

`@end example`
: for pre-formatted text which is indented and typeset in typewriter font.

Example:

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

↦

before example

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

after example

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

`@format` ...

`@end format`

for pre-formatted text which is not indented and typeset in normal font.

Example:

```
before format
@format
in              format
no extra identation but still
escape of special characters like @{,@},@@
@end format
after format
```

↦

before format

in format
no extra identation but still
escape of special characters like `{`,`}`,`@`

after format

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

`@texinfo` ...

`@end texinfo`

for text which is written in pure texinfo.

Example:

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

↦

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

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

See also Section 3.9.6 [template_lib], page 59 for an examples of the typesetting rules explained here.

3.9.4 The help string of a library

The help (or, info) string of a library should have the following format:

```
info="
LIBRARY: <library_name> <one line description of the content>
AUTHOR:  <name, and email address of author>
[SEE ALSO: <comma-separated words of cross references>]
[KEYWORDS: <semicolon-separated phrases of index keys>]
PROCEDURES:
  <procedure1>;     <one line description of the purpose>
   .
   .
  <procedureN>;     <one line description of the purpose>
";
```

Only such procedures should be listed in the `PROCEDURE` section which are not `static` and which have a help and example section.

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. If this description consists of only upper-case characters, then it will be typeset in all lowercase characters in the manual.

For more information, see Section 3.9.3 [Typesetting of help strings], page 56. For an example, see Section 3.9.6 [template_lib], page 59.

3.9.5 The help string of procedures

The help string of a procedure should have the following format:

```
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>]
[SEE ALSO: <comma-separated names of related procedures/cross references>]
[KEYWORDS: <semicolon-separated phrases of index keys>]
EXAMPLE:  example <proc_name>; shows an example
```

Further arbitrary items (like `THEORY:`, or `BACKGROUND:`) are recognized, as well, but should be used diligently.

Remember that help strings are formatted as explained in Section 3.9.3 [Typesetting of help strings], page 56. In particular, descriptions may contain the texinfo markup elements `@*`, `@math{..}`, `@code{..}`, `@example`, `@format`, `@texinfo` to better control their typesetting. See Section 3.9.6.3 [msum], page 63, Section 3.9.6.1 [mdouble], page 62, Section 3.9.6.2 [mtripple], page 62 for examples.

3.9.6 template_lib

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

```
////////////////////////////////////////////////////////////////////
// version string automatically expanded by CVS
```

```
version="$Id: template.lib,v 1.10 2001/01/16 13:48:46 Singular Exp $";
category="Miscellaneous";
// summary description of the library
info="
LIBRARY:   template.lib  A Template for a Singular Library
AUTHOR:    Olaf Bachmann, email: obachman@mathematik.uni-kl.de

SEE ALSO:  standard_lib, Guidelines for writing a library,
           Typesetting of help strings

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

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

@item @strong{Return:}
int: @math{i+i+i}
@item @strong{Note:}
Help is in pure Texinfo
@*This help string is written in texinfo, which enables you to use,
among others, the @@math command for mathematical typesetting (like
@math{\alpha, \beta}).
@*It also gives more control over the layout, but is, admittingly,
more cumbersome to write.
@end table
@c use @c ref contstuct for references
```

```
@cindex procedure, texinfo help
@c ref
@strong{See also:}
@ref{mdouble}, @ref{msum}, @ref{Typesetting of help strings}
@c ref
"
{
  return (i + i + i);
}
example
{ "EXAMPLE:"; echo = 2;
  mtripple(0);
  mtripple(-1);
}
////////////////////////////////////////////////////////////////////
proc msum(list #)
"USAGE:  msum([i_1,..,i_n]); @code{i_1,..,i_n} def
RETURN:  Sum of int arguments
NOTE:    This help string is written in a mixture of ASCII and texinfo
         @* Use a @ref constructs for references (like @pxref{mtripple})
         @* Use @code  for typewriter font (like @code{i_1})
         @* Use @math  for simple math mode typesetting (like @math{i_1}).
         @* Note: No parenthesis like } are allowed inside @math and @code
         @* Use @example for indented preformatted text typeset in typewriter
         font like
@example
         this  --> that
@end example
        Use @format  for preformatted text typeset in normal font
@format
         this --> that
@end format
        Use @texinfo for text in pure texinfo
@texinfo
@expansion{}
@tex
$i_{1,1}$
@end tex

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

```
  }
  return (s);
}
example
{ "EXAMPLE:"; echo = 2;
  msum();
  msum(4);
  msum(1,2,3,4);
}
```

After typesetting, the library appears in the document as follows (with one subsection for each procedure):

Library: template.lib

Purpose: A Template for a Singular Library

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

Procedures: **See also:** Section 3.9 [Guidelines for writing a library], page 53; Section 3.9.3 [Typesetting of help strings], page 56; Section D.1 [standard_lib], page 293.

3.9.6.1 mdouble

Procedure from library `template.lib` (see Section 3.9.6 [template_lib], page 59).

Usage: mdouble(i); i int

Return: int: i+i

Note: Help string is in pure ASCII
this line starts on a new line since previous line is short mdouble(i): no new line

Example:

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

See also: Section 3.9.3 [Typesetting of help strings], page 56; Section 3.9.6.3 [msum], page 63; Section 3.9.6.2 [mtripple], page 62.

3.9.6.2 mtripple

Procedure from library `template.lib` (see Section 3.9.6 [template_lib], page 59).

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

Return: int: $i + i + i$

Note: Help is in pure Texinfo
This help string is written in texinfo, which enables you to use, among others, the `@math` command for mathematical typesetting (like

α, β).
It also gives more control over the layout, but is, admittingly, more cumbersome to write.

See also: Section 3.9.3 [Typesetting of help strings], page 56; Section 3.9.6.1 [mdouble], page 62; Section 3.9.6.3 [msum], page 63.

Example:

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

3.9.6.3 msum

Procedure from library `template.lib` (see Section 3.9.6 [template_lib], page 59).

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

Return: Sum of int arguments

Note: This help string is written in a mixture of ASCII and texinfo
Use a `@ref` constructs for references (like see Section 3.9.6.2 [mtripple], page 62)
Use `@code` for typewriter font (like `i_1`)
Use `@math` for simple math mode typesetting (like i_1).
Note: No parenthesis like `}` are allowed inside `@math` and `@code`
Use `@example` for indented preformatted text typeset in typewriter font like

```
this  --> that
```

Use `@format` for preformatted text typeset in normal font

this –> that

Use `@texinfo` for text in pure texinfo

↦ $i_{1,1}$

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

Example:

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

See also: Section 3.9.3 [Typesetting of help strings], page 56; Section 3.9.6.1 [mdouble], page 62; Section 3.9.6.2 [mtripple], page 62.

3.10 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⟩ .

3.10.1 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 223.

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)
```

3.10.2 Source code debugger

The source code debugger (sdb) is an experimental feature, it's interface may change in future versions of Singular.
To enable the use of the source code debugger SINGULAR has to be started with the option `-d` or `--sdb` (see Section 3.1.6 [Command line options], page 18).

sdb commands

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

`b` print backtrace of calling stack

`c` continue

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

`h`,? display help screen

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

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

`Q` quit this Singular session

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

Syntactical errors in procedures

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

SDB breakpoints in procedures

Up to seven SDB breakpoints can be set. To set a breakpoint at a procedure use `breakpoint`. (See Section 5.2.2 [breakpoint], page 215).
These breakpoints can be cleared with the command `d breakpoint_no` from within the debugger or with `breakpoint(` proc_name `,-1);`.

3.10.3 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 5.2.12 [~], page 219.

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
```

3.10.4 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 130; Section 5.3.11 [voice], page 224.

3.10.5 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 procedure body '`{`'
- eob: end of procedure body '`}`'
- 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 3.5.1 [General command syntax], page 38.

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 (possibly unnamed) ring. This means that entries in one of these rings appear also in the other ones. The following defines a ring `s` which is just another reference (or name) for the basering `r`.

```
  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
  ring t=32003,(u,w),dp;
  def rt=r+t;
  rt;
↦ //   characteristic : 32003
↦ //   number of vars : 5
↦ //        block   1 : ordering dp
↦ //                  : names    x y z
↦ //        block   2 : ordering dp
↦ //                  : names    u w
↦ //        block   3 : ordering C
```

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

4.1.1 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 5.1.129 [typeof], page 209.

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_and_ideal_expressions ;
`ideal` name = ideal_expression ;

Purpose: defines an ideal.

Default: 0

Example:

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

4.2.2 ideal expressions

An ideal expression is:

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

5. a type cast to ideal

Example:

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

4.2.3 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 5.1.114 [simplify], page 196.

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 126)

`coeffs`
: matrix of coefficients (see Section 5.1.10 [coeffs], page 128)

`contract`
: contraction by an ideal (see Section 5.1.11 [contract], page 130)

`diff`
: partial derivative (see Section 5.1.18 [diff], page 133)

`degree`
: multiplicity, dimension and codimension of the ideal of leading terms (see Section 5.1.15 [degree], page 132)

`dim`
: Krull dimension of basering modulo the ideal of leading terms (see Section 5.1.19 [dim], page 134)

`eliminate`
: elimination of variables (see Section 5.1.22 [eliminate], page 136)

`facstd` factorizing Groebner basis algorithm (see Section 5.1.29 [facstd], page 139)

`factorize`
ideal of factors of a polynomial (see Section 5.1.30 [factorize], page 140)

`fglm` Groebner basis computation from a Groebner basis w.r.t. a different ordering (see Section 5.1.32 [fglm], page 142)

`finduni` computation of univariate polynomials lying in a zero dimensional ideal (see Section 5.1.36 [finduni], page 143)

`groebner` Groebner basis computation (a wrapper around `std,stdhilb,stdfglm`,...) (see Section 5.1.42 [groebner], page 147)

`highcorner`
computes the smallest monomial not contained in the ideal. The ideal has to be zero-dimensional. (see Section 5.1.44 [highcorner], page 149)

`homog` homogenization with respect to a variable (see Section 5.1.46 [homog], page 150)

`hilb` Hilbert series of a standard basis (see Section 5.1.45 [hilb], page 149)

`indepSet` sets of independent variables of an ideal (see Section 5.1.50 [indepSet], page 153)

`interred` interreduction of an ideal (see Section 5.1.52 [interred], page 154)

`intersect`
ideal intersection (see Section 5.1.53 [intersect], page 155)

`jacob` ideal of all partial derivatives resp. jacobian matrix (see Section 5.1.54 [jacob], page 155)

`jet` Taylor series up to a given order (see Section 5.1.55 [jet], page 156)

`kbase` vector space basis of basering modulo ideal of leading terms (see Section 5.1.56 [kbase], page 157)

`koszul` Koszul matrix (see Section 5.1.59 [koszul], page 159)

`lead` leading terms of a set of generators (see Section 5.1.61 [lead], page 160)

`lift` lift-matrix (see Section 5.1.66 [lift], page 163)

`liftstd` standard basis and transformation matrix computation (see Section 5.1.67 [liftstd], page 163)

`lres` free resolution for homogeneous ideals (see Section 5.1.69 [lres], page 165)

`maxideal` power of the maximal ideal at 0 (see Section 5.1.70 [maxideal], page 166)

`minbase` minimal generating set of a homogeneous ideal, resp. module, or an ideal, resp. module, in a local ring (see Section 5.1.72 [minbase], page 167)

`minor` set of minors of a matrix (see Section 5.1.73 [minor], page 168)

`modulo` represents $(h1+h2)/h1 \cong h2/(h1 \cap h2)$ (see Section 5.1.75 [modulo], page 169)

`mres` minimal free resolution of an ideal resp. module w.r.t. a minimal set of generators of the given ideal resp. module (see Section 5.1.78 [mres], page 171)

`mstd` standard basis and minimal generating set of an ideal (see Section 5.1.79 [mstd], page 171)

`mult` multiplicity, resp. degree, of the ideal of leading terms (see Section 5.1.80 [mult], page 172)

ncols number of columns (see Section 5.1.83 [ncols], page 174)

preimage preimage under a ring map (see Section 5.1.95 [preimage], page 182)

qhweight quasihomogeneous weights of an ideal (see Section 5.1.100 [qhweight], page 187)

quotient ideal quotient (see Section 5.1.103 [quotient], page 189)

reduce normalform with respect to a standard basis (see Section 5.1.106 [reduce], page 191)

res free resolution of an ideal resp. module but not changing the given ideal resp. module (see Section 5.1.109 [res], page 193)

simplify simplify a set of polynomials (see Section 5.1.114 [simplify], page 196)

size number of non-zero generators (see Section 5.1.115 [size], page 197)

sortvec permutation for sorting ideals resp. modules (see Section 5.1.116 [sortvec], page 198)

sres free resolution of a standard basis (see Section 5.1.118 [sres], page 200)

std standard basis computation (see Section 5.1.120 [std], page 202)

stdfglm standard basis computation with fglm technique (see Section 5.1.121 [stdfglm], page 204)

stdhilb Hilbert driven standard basis computation (see Section 5.1.122 [stdhilb], page 204

subst substitute a ring variable (see Section 5.1.123 [subst], page 205)

syz computation of the first syzygy module (see Section 5.1.125 [syz], page 207)

vdim vector space dimension of basering modulo ideal of leading terms (see Section 5.1.134 [vdim], page 212)

weight optimal weights (see Section 5.1.136 [weight], page 212)

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 5.2.6 [for], page 216), and to represent boolean values (FALSE is represented by 0, every other value means TRUE, see Section 4.3.5 [boolean expressions], page 76).

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 225). If this range is too small the expression must be converted to the type number over a ring with characteristic 0.

Example:

```
12345678901; // too large
↦    ? ‘12345678901‘ greater than 2147483647(max. integer representation)
↦    ? error occurred in line 1: ‘12345678901; // too large‘
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 37 in the actual coefficient field, here it equals 6
g;
↦ 6
```

See Section 3.5.5 [Type conversion and casting], page 42; Section 4.11 [number], page 103.

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

`/` integer division (omitting the remainder), rounding toward 0

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

`%` integer modulo (the remainder of the division `/`)

`mod` integer modulo (the remainder of the division `div`), always non-negative

`^`, `**` 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 225.

Example:

```
  int i=1;
  int j;
  i++; i;  i--; i;
↦ 2
↦ 1
  // ++ and -- do not return a value as in C, can not assign
  j = i++;
↦ // ** right side is not a datum, assignment ignored
  // the value of j is unchanged
  j; i;
↦ 0
↦ 2
  i+2, 2-i, 5^2;
↦ 4 0 25
  5 div 2, 8%3;
↦ 2 2
  -5 div 2, -5 / 2, -5 mod 2, -5 % 2;
↦ -3 -2 1 -1
  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 125)

deg degree of a poly resp. vector (see Section 5.1.14 [deg], page 131)

det determinant (see Section 5.1.17 [det], page 133)

dim Krull dimension of basering modulo ideal of leading terms, resp. dimension of module of leading terms (see Section 5.1.19 [dim], page 134)

extgcd Bezout representation of gcd (see Section 5.1.28 [extgcd], page 139)

find position of a substring in a string (see Section 5.1.35 [find], page 143)

gcd greatest common divisor (see Section 5.1.39 [gcd], page 146)

koszul Koszul matrix (see Section 5.1.59 [koszul], page 159)

memory memory usage (see Section 5.1.71 [memory], page 166)

mult multiplicity of an ideal, resp. module, of leading terms (see Section 5.1.80 [mult], page 172)

ncols number of columns (see Section 5.1.83 [ncols], page 174)

npars number of ring parameters (see Section 5.1.84 [npars], page 175)

nrows number of rows of a matrix, resp. the rank of the free module where the vector or module lives (see Section 5.1.86 [nrows], page 176)

nvars number of ring variables (see Section 5.1.87 [nvars], page 176)

ord degree of the leading term of a poly resp. vector (see Section 5.1.90 [ord], page 180)

par n-th parameter of the basering (see Section 5.1.92 [par], page 181)

pardeg degree of a number considered as a polynomial in the ring parameters (see Section 5.1.93 [pardeg], page 182) `prime` the next lower prime (see Section 5.1.96 [prime], page 183)

`random` a pseudo random integer between the given limits (see Section 5.1.104 [random], page 190)

`regularity`
regularity of a resolution (see Section 5.1.107 [regularity], page 192)

`rvar` test, if the given expression or string is a ring variable (see Section 5.1.112 [rvar], page 195)

`size` number of elements in an object (see Section 5.1.115 [size], page 197)

`trace` trace of an integer matrix (see Section 5.1.126 [trace], page 208)

`var` n-th ring variable of the basering (see Section 5.1.132 [var], page 211)

`vdim` vector space dimension of basering modulo ideal of leading terms, resp. of freemodule modulo module of leading terms (see Section 5.1.134 [vdim], page 212)

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 the 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 is always the smallest element,
   - in Z/p the representatives in the range -(p-1)/2..(p-1)/2 when p>2 resp. 0 and 1 for p=2 are used for comparisons,
   - in field extensions the last two operations (`>=,<=`) yield always TRUE (1) and the `<` and `>` are equivalent to `!=`.

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

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

See Section 6.2 [Major differences to the C programming language], page 225.

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 225. 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_and_intvec_and_intmat_expressions ;
rows and cols must be positive int expressions.

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

Default: 0 (1 x 1 matrix)

Example:

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

4.4.2 intmat expressions

An intmat expression is:

1. an identifier of type intmat
2. a function returning intmat
3. intmat operations with int (`+`, `-`, `*`, `div`, `%`)
4. intmat operations (`+`, `-`, `*`)
5. a type cast to intmat (see Section 4.4.3 [intmat type cast], page 79)

Example:

```
  intmat Idm[2][2];
  Idm +1;          // add the unit intmat
↦ 1,0,
↦ 0,1
  intmat m1[3][2] = _,1,-2;  // take entries from the last result
```

```
  m1;
↦ 1,0,
↦ 0,1,
↦ 1,-2
  intmat m2[2][3]=1,0,2,4,5,1;
  transpose(m2);
↦ 1,4,
↦ 0,5,
↦ 2,1
  intvec v1=1,2,4;
  intvec v2=5,7,8;
  m1=v1,v2;         // fill m1 with v1 and v2
  m1;
↦ 1,2,
↦ 4,5,
↦ 7,8
  trace(m1*m2);
↦ 56
```

See Section 3.5.5 [Type conversion and casting], page 42; Section 4.11 [number], page 103.

4.4.3 intmat type cast

Syntax: `intmat` (expression)
`intmat` (expression, int_n, int_m)

Type: intmat

Purpose: Converts expression to an intmat, where expression must be of type intvec, or intmat. If int_n and int_m are supplied, then they specify the dimension of the intmat. Otherwise, the size (resp. dimensions) of the intmat are determined by the size (resp. dimensions) of the expression.

Example:

```
  intmat(intvec(1));
↦ 1
  intmat(intvec(1), 1, 2);
↦ 1,0
  intmat(intvec(1,2,3,4), 2, 2);
↦ 1,2,
↦ 3,4
  intmat(_, 2, 3);
↦ 1,2,3,
↦ 4,0,0
  intmat(_, 2, 1);
↦ 1,
↦ 2
```

See Section 3.5.5 [Type conversion and casting], page 42; Section 4.4 [intmat], page 78; Section 4.9.3 [matrix type cast], page 98.

4.4.4 intmat operations

`+` addition with intmat or int; the int is converted into a diagonal intmat

`-` negation or subtraction with intmat or int; the int is converted into a diagonal intmat

`*` multiplication with intmat, intvec, or int; the int is converted into a diagonal intmat

`div,/` division of entries in the integers (omitting the remainder)

`%, mod` entries modulo int (remainder of the division)

`<>, ==` comparison

intmat_expression `[` intvec_expression`,` intvec_expression `]`
is an intmat entry, where the first index indicates the row and the second the column

Example:

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

4.4.5 intmat related functions

`betti` Betti numbers of a free resolution (see Section 5.1.3 [betti], page 124)

`det` determinant (see Section 5.1.17 [det], page 133) `ncols` number of cols (see Section 5.1.83 [ncols], page 174)

`nrows` number of rows (see Section 5.1.86 [nrows], page 176)

`random` create a pseudo random intmat (see Section 5.1.104 [random], page 190)

`size` total number of entries (see Section 5.1.115 [size], page 197)

`transpose`
transpose of an intmat (see Section 5.1.127 [transpose], page 208)

`trace` trace of an intmat (see Section 5.1.126 [trace], page 208)

4.5 intvec

Variables of type intvec are lists of integers. For the range of integers see Section 6.1 [Limitations], page 225. They may be used for simulating sets of integers (and other sets if the intvec is used as an index set for other objects). Addition and subtraction of an intvec with an int or an intvec is done element-wise.

4.5.1 intvec declarations

Syntax: `intvec` name = intvec_expression ;
`intvec` name = list_of_int_and_intvec_expressions ;

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

Default: 0

Example:

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

4.5.2 intvec expressions

An intvec expression is:

1. a range: int expression `..` int expression
2. a function returning intvec
3. intvec operations with int (`+`, `-`, `*`, `/`, `%`)
4. intvec operations (`+`, `-`)
5. intvec operation with intmat (`*`)
6. 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 by 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;              // component-wise addition
↦ 2,6,10,14,16,0
```

```
  iv[size(iv)-1];      // last-1 entry
↦ 8
  intvec iw=2,3,4,0;
  iv==iw;              // lexicographic comparison
↦ 0
  iv < iw;
↦ 1
  iv != iw;
↦ 1
  iv[2];
↦ 3
  iw = 4,1,2;
  iv[iw];
↦ 7 1 3
```

4.5.4 intvec related functions

`hilb` returns Hilbert series as intvec (see Section 5.1.45 [hilb], page 149)

`indepSet` sets of independent variables of an ideal (see Section 5.1.50 [indepSet], page 153)

`leadexp` the exponent vector of the leading monomial (see Section 5.1.63 [leadexp], page 161)

`nrows` number of rows (see Section 5.1.86 [nrows], page 176)

`qhweight` returns quasihomogeneous weights (see Section 5.1.100 [qhweight], page 187)

`size` length of the intvec (see Section 5.1.115 [size], page 197)

`sortvec` permutation for sorting ideals/modules (see Section 5.1.116 [sortvec], page 198)

`transpose`
transpose of an intvec, returns an intmat (see Section 5.1.127 [transpose], page 208)

`weight` returns weights for the weighted ecart method (see Section 5.1.136 [weight], page 212)

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 85)
- MPfile links (see Section 4.6.5.1 [MPfile links], page 86)
- MPtcp links (see Section 4.6.5.2 [MPtcp links], page 88)
- DBM links (see Section 4.6.6 [DBM links], page 90)

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 85
- for MPfile links: Section 4.6.5.1 [MPfile links], page 86
- for MPtcp links: Section 4.6.5.2 [MPtcp links], page 88
- for DBM links: Section 4.6.6 [DBM links], page 90

4.6.3 link related functions

`close` closes a link (see Section 5.1.8 [close], page 127)

`dump` generates a dump of all variables and their values (see Section 5.1.21 [dump], page 135)

`getdump` reads a dump (see Section 5.1.41 [getdump], page 146)

`open` opens a link (see Section 5.1.88 [open], page 177)

`read` reads from a link (see Section 5.1.105 [read], page 190)

`status` gets the status of a link (see Section 5.1.119 [status], page 201)

`write` writes to a link (see Section 5.1.137 [write], page 213)

`kill` closes and kills a link (see Section 5.1.57 [kill], page 158)

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 222). 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 5.1.26 [execute], page 138).

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

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

There are the following default values:

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

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

Note that the filename may contain a path. On Microsoft Windows (resp. MS-DOS) platforms, names of a drive can precede the filename, but must be started with a `//` (as in `//c/temp/ex`. An ASCII link can be used either for reading or for writing, but not for both at the same time. A `close` command must be used before a change of I/O direction. Types without a conversion to `string` cannot be written.

Example:

```
  ring r=32003,(x,y,z),dp;
  link l=":w example.txt";     // type is ASCII, mode is overwrite
  l;
↦ // type : ASCII
↦ // mode : w
↦ // name : example.txt
↦ // open : no
↦ // read : not ready
↦ // write: not ready
  status(l, "open", "yes");    // link is not yet opened
↦ 0
  ideal i=x2,y2,z2;
  write (l,1,";",2,";","ideal i=",i,";");
  status(l, "open", "yes");    // now link is open
↦ 1
```

```
  status(l, "mode");           // for writing
↦ w
  close(l);                    // link is closed
  write("example.txt","int j=5;");// data is appended to file
  read("example.txt");         // data is returned as string
↦ 1
↦ ;
↦ 2
↦ ;
↦ ideal i=
↦ x2,y2,z2
↦ ;
↦ int j=5;
↦ 
  execute(read(l));            // read string is executed
↦ 1
↦ 2
↦ // ** redefining i **
  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 cannot 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 MPfile 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 --MPrsh " +` rsh
  6. `"MPtcp:launch --MPrsh " +` rsh `+ " --MPhost " +` hostname
  7. `"MPtcp:launch --MPrsh " +` rsh `+ " --MPhost " +` hostname `+ " --MPapplication " +` application

  SINGULAR becomes a client and starts (launches) the application using the specified remote shell command (default is `ssh`) on a (possibly) different host (default is `localhost` which then 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 remote shell (rsh) command is specified, then the command `ssh` is used.
- if no application is specified (in mode `launch`) the default application is the value of `system("Singular") + "-bq"`. (This evaluates to the absolute path of the SINGULAR currently running with the option `"-bq"` 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 a remote shell command, 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 are to 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 5.1.102 [quote], page 188), 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 everything, 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`), which contain the data base, are automatically created.

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

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

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

Example:

```
  link l="DBM:rw example";
  write(l,"1","abc");
  write(l,"3","XYZ");
```

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

4.7 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. an identifier of type list
3. a function returning list
4. list expressions combined by the arithmetic operation `+`
5. a type cast to list

See Section 3.5.5 [Type conversion and casting], page 42.

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 122)

`betti` Betti numbers of a resolution (see Section 5.1.3 [betti], page 124)

`delete` deletes an element from a list (see Section 5.1.16 [delete], page 132)

`facstd` factorizing Groebner basis algorithm (see Section 5.1.29 [facstd], page 139)

`factorize`
list of factors of a polynomial (see Section 5.1.30 [factorize], page 140)

`insert` inserts or appends a new element to a list (see Section 5.1.51 [insert], page 154)

`lres` free resolution (see Section 5.1.69 [lres], page 165)

`minres` minimize a free resolution (see Section 5.1.74 [minres], page 168)

`mres` minimal free resolution of an ideal, resp. module w.r.t. a minimal set of generators of the first module (see Section 5.1.78 [mres], page 171)

`names` list of all user-defined variable names (see Section 5.1.82 [names], page 173)

`res` free resolution of an ideal, resp. module (see Section 5.1.109 [res], page 193)

`size` number of entries (see Section 5.1.115 [size], page 197)

`sres` free resolution of an ideal, resp. module, given by a standard base (see Section 5.1.118 [sres], page 200)

4.8 map

Maps are ring maps from a preimage ring into the basering.

Note:

- the target of a map is **ALWAYS** the actual basering
- the preimage ring is stored "by name", that means, maps can only be used in such contexts, where the name of the preimage ring can be resolved (i.e., there might be problems for rings/maps defined in subprocedures). See also Section 6.4 [Identifier resolution], page 229, Section 3.7.2 [Names in procedures], page 49.

Maps between rings with different coefficient fields are possible and listed below.

Canonically realized are

- $Q \rightarrow Q(a, \ldots)$
- $Q \rightarrow R$
- $Q \rightarrow C$
- $Z/p \rightarrow (Z/p)(a, \ldots)$
- $Z/p \rightarrow GF(p^n)$
- $Z/p \rightarrow R$
- $R \rightarrow C$

Possible are furthermore

- $Z/p \to Q, \quad [i]_p \mapsto i \in [-p/2,\, p/2] \subseteq Z$
- $Z/p \to Z/p', \quad [i]_p \mapsto i \in [-p/2,\, p/2] \subseteq Z,\ i \mapsto [i]_{p'} \in Z/p'$
- $C \to R, \quad$ the real part

Finally, in Singular we allow the mapping from rings with coefficient field Q to rings whose ground fields have finite characteristic:

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

In these cases the denominator and the numerator of a number are mapped separately by the usual map from Z to Z/p, and the image of the number is built again afterwards by division. It is thus not allowed to map numbers whose denominator is divisible by the characteristic of the target ground field, or objects containing such numbers. We, therefore, strongly recommend using such maps only to map objects with integer coefficients.

See Section 5.1.48 [imap], page 152; Section 5.1.31 [fetch], page 141; Section 5.1.123 [subst], page 205.

4.8.1 map declarations

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

Purpose: defines a ring map from preimage_ring to basering.
Maps the variables of the preimage ring to the generators of the ideal. If the ideal contains less elements than variables in the preimage_ring the remaining variables are mapped to 0, if the ideal contains more elements these are ignored. The image ring is always the actual basering. For the mapping of coefficients from different fields see Section 4.8 [map], page 94.

Default: none

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

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

Example:

```
  ring r1=32003,(x,y,z),dp;
  ideal i=x,y,z;
  ring r2=32003,(a,b),dp;
  map f=r1,a,b,a+b;
  // maps from r1 to r2,
  // x -> a
  // y -> b
  // z -> a+b
  f(i);
↦ _[1]=a
↦ _[2]=b
↦ _[3]=a+b
  // operations like f(i[1]) or f(i*i) are not allowed
  ideal i=f(i);
```

```
  // objects in different rings may have the same name
  map g   = r2,a2,b2;
  map phi = g(f);
  // composition of map f and g
  // maps from r1 to r2,
  // x -> a2
  // y -> b2
  // z -> a2+b2
  phi(i);
↦ _[1]=a2
↦ _[2]=b2
↦ _[3]=a2+b2
```

See Section 5.1.31 [fetch], page 141; Section 4.2.2 [ideal expressions], page 68; Section 5.1.48 [imap], page 152; Section 4.8 [map], page 94; Section 4.16 [ring], page 113.

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 a 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 is the identity
  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 gets the same size as the right-hand side, otherwise the size is determined by the left-hand side. If the size is omitted a 1x1 matrix is created.

Default: 0 (1 x 1 matrix)

Example:

```
  int ro = 3;
  ring r = 32003,(x,y,z),dp;
  poly f=xyz;
  poly g=z*f;
  ideal i=f,g,g^2;
  matrix m[ro][3] = x3y4, 0, i, f ; // a 3 x 3 matrix
  m;
↦ m[1,1]=x3y4
↦ m[1,2]=0
↦ m[1,3]=xyz
↦ m[2,1]=xyz2
↦ m[2,2]=x2y2z4
↦ m[2,3]=xyz
↦ m[3,1]=0
↦ m[3,2]=0
↦ m[3,3]=0
  print(m);
↦ x3y4,0,     xyz,
↦ xyz2,x2y2z4,xyz,
↦ 0,   0,     0
  matrix A;   // the 1 x 1 zero matrix
  matrix B[2][2] = m[1..2, 2..3]; //defines a submatrix
  print(B);
↦ 0,     xyz,
↦ x2y2z4,xyz
  matrix C=m; // defines C as a 3 x 3 matrix equal to m
  print(C);
↦ x3y4,0,     xyz,
↦ xyz2,x2y2z4,xyz,
↦ 0,   0,     0
```

4.9.2 matrix expressions

A matrix expression is:

1. an identifier of type matrix
2. a function returning matrix
3. matrix expressions combined by the arithmetic operations +, - or *
4. a type cast to matrix (see Section 4.9.3 [matrix type cast], page 98)

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 type cast

Syntax: `matrix` (expression)
`matrix` (expression, int_n, int_m)

Type: matrix

Purpose: Converts expression to a matrix, where expression must be of type int, intmat, intvec, number, poly, ideal, vector, module, or matrix. If int_n and int_m are supplied, then they specify the dimension of the matrix. Otherwise, the size (resp. dimensions) of the matrix are determined by the size (resp. dimensions) of the expression.

Example:

```
  ring r=32003,(x,y,z),dp;
  matrix(x);
↦ _[1,1]=x
  matrix(x, 1, 2);
↦ _[1,1]=x
↦ _[1,2]=0
  matrix(intmat(intvec(1,2,3,4), 2, 2));
↦ _[1,1]=1
↦ _[1,2]=2
↦ _[2,1]=3
↦ _[2,2]=4
  matrix(_, 2, 3);
↦ _[1,1]=1
↦ _[1,2]=2
↦ _[1,3]=0
↦ _[2,1]=3
```

```
↦ _[2,2]=4
↦ _[2,3]=0
  matrix(_, 2, 1);
↦ _[1,1]=1
↦ _[2,1]=3
```

See Section 3.5.5 [Type conversion and casting], page 42; Section 4.4.3 [intmat type cast], page 79; Section 4.9 [matrix], page 96.

4.9.4 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

`/` division by poly

`==`, `<>`, `!=` 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; // define a matrix
  print(A); // nice printing of small matrices
↦ 1,3,2,
↦ 5,0,3,
↦ 2,4,5
  A[2,3];   // matrix entry
↦ 3
  A[2,3] = A[2,3] + 1; // change entry
  A[2,1..3] = 1,2,3;   // change 2nd row
  print(A);
↦ 1,3,2,
↦ 1,2,3,
↦ 2,4,5
  matrix E[3][3]; E = E + 1;  // the unit matrix
  matrix B =x*E - A;
  print(B);
↦ x-1,-3, -2,
↦ -1, x-2,-3,
↦ -2, -4, x-5
  det(B);        // the characteristic polynomial of A
↦ x3-8x2-2x-1
  A*A*A - 8 * A*A - 2*A == E;  // Cayley-Hamilton
↦ 1
  vector v =[x,-1,x2];
  A*v; // multiplication of matrix and vector
↦ _[1,1]=2x2+x-3
↦ _[2,1]=3x2+x-2
↦ _[3,1]=5x2+2x-4
```

```
  matrix m[2][2]=1,2,3;
  print(m-transpose(m));
↦ 0,-1,
↦ 1,0
```

4.9.5 matrix related functions

`bareiss` Gauss-Bareiss algorithm (see Section 5.1.2 [bareiss], page 122)

`coef` matrix of coefficients and monomials (see Section 5.1.9 [coef], page 127)

`coeffs` matrix of coefficients (see Section 5.1.10 [coeffs], page 128)

`det` determinant (see Section 5.1.17 [det], page 133)

`diff` partial derivative (see Section 5.1.18 [diff], page 133)

`jacob` Jacobi matrix (see Section 5.1.54 [jacob], page 155)

`koszul` Koszul matrix (see Section 5.1.59 [koszul], page 159)

`lift` lift-matrix (see Section 5.1.66 [lift], page 163)

`liftstd` standard basis and transformation matrix computation (see Section 5.1.67 [liftstd], page 163)

`minor` set of minors of a matrix (see Section 5.1.73 [minor], page 168)

`ncols` number of columns (see Section 5.1.83 [ncols], page 174)

`nrows` number of rows (see Section 5.1.86 [nrows], page 176)

`print` nice print format (see Section 5.1.97 [print], page 183)

`size` number of matrix entries (see Section 5.1.115 [size], page 197)

`subst` substitute a ring variable (see Section 5.1.123 [subst], page 205)

`trace` trace of a matrix (see Section 5.1.126 [trace], page 208)

`transpose`
transpose a matrix (see Section 5.1.127 [transpose], page 208)

`wedge` wedge product (see Section 5.1.135 [wedge], page 212)

See also the library Section D.3.1 [matrix_lib], page 338, which contains more matrix-related functions.

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;` create the presentation matrix of size n×k for R^n/M, i.e., the columns of A are the vectors $v_1, \ldots, v_k$ which generate M (cf. Section B.1 [Representation of mathematical objects], page 274).

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

See Section 3.5.5 [Type conversion and casting], page 42; Section 4.2 [ideal], page 68; Section 4.12 [poly], page 107; Section 4.18 [vector], page 118.

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 128)

`degree` multiplicity, dimension and codimension of the module of leading terms (see Section 5.1.15 [degree], page 132)

`diff` partial derivative (see Section 5.1.18 [diff], page 133)

`dim` Krull dimension of free module over the basering modulo the module of leading terms (see Section 5.1.19 [dim], page 134)

`eliminate`
elimination of variables (see Section 5.1.22 [eliminate], page 136)

`freemodule`
the free module of given rank (see Section 5.1.38 [freemodule], page 145)

`groebner` Groebner basis computation (a wrapper around `std,stdhilb,stdfglm`,...) (see Section 5.1.42 [groebner], page 147)

`hilb` Hilbert function of a standard basis (see Section 5.1.45 [hilb], page 149)

`homog` homogenization with respect to a variable (see Section 5.1.46 [homog], page 150)

`interred` interreduction of a module (see Section 5.1.52 [interred], page 154)

`intersect`
module intersection (see Section 5.1.53 [intersect], page 155)

`jet` Taylor series up to a given order (see Section 5.1.55 [jet], page 156)

`kbase` vector space basis of free module over the basering modulo the module of leading terms (see Section 5.1.56 [kbase], page 157)

`lead` initial module (see Section 5.1.61 [lead], page 160)

`lift` lift-matrix (see Section 5.1.66 [lift], page 163)

`liftstd` standard basis and transformation matrix computation (see Section 5.1.67 [liftstd], page 163)

`lres` free resolution (see Section 5.1.69 [lres], page 165)

`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 5.1.75 [modulo], page 169)

`mres` minimal free resolution of an ideal resp. module w.r.t. a minimal set of generators of the given module (see Section 5.1.78 [mres], page 171)

`mult` multiplicity, resp. degree, of the module of leading terms (see Section 5.1.80 [mult], page 172)

`ncols` number of columns (see Section 5.1.83 [ncols], page 174)

`nrows` number of rows (see Section 5.1.86 [nrows], page 176) `print` nice print format (see Section 5.1.97 [print], page 183)

`prune` minimize the embedding into a free module (see Section 5.1.99 [prune], page 186)

`qhweight` quasihomogeneous weights of an ideal, resp. module (see Section 5.1.100 [qhweight], page 187)

`quotient` module quotient (see Section 5.1.103 [quotient], page 189)

`reduce` normalform with respect to a standard basis (see Section 5.1.106 [reduce], page 191)

`res` free resolution of an ideal, resp. module, but not changing the given ideal, resp. module (see Section 5.1.109 [res], page 193)

`simplify` simplify a set of vectors (see Section 5.1.114 [simplify], page 196)

`size` number of non-zero generators (see Section 5.1.115 [size], page 197)

`sortvec` permutation for sorting ideals/modules (see Section 5.1.116 [sortvec], page 198)

`sres` free resolution of a standard basis (see Section 5.1.118 [sres], page 200)

`std` standard basis computation (see Section 5.1.120 [std], page 202, Section 5.1.67 [liftstd], page 163)

`subst` substitute a ring variable (see Section 5.1.123 [subst], page 205)

`syz` computation of the first syzygy module (see Section 5.1.125 [syz], page 207)

`vdim` vector space dimension of free module over the basering modulo module of leading terms (see Section 5.1.134 [vdim], page 212)

`weight` "optimal" weights (see Section 5.1.136 [weight], page 212)

4.11 number

Numbers are elements from the coefficient field (or ground field). They can only be defined or accessed with respect to a basering which determines the coefficient field. See Section 4.16.1 [ring declarations], page 113 for declarations of coefficient fields.

Warning: Beware of the special meaning of the letter `e` (immediately following a sequence of digits) if the field is real (or complex). See Section 4.11.3 [number operations], page 105, Section 6.3 [Miscellaneous oddities], page 228.

4.11.1 number declarations

Syntax: `number` name = number_expression ;

Purpose: defines a number.

Default: 0

Note: Numbers may only be declared w.r.t. the coefficient field of the current basering, i.e., a ring has to be defined prior to any number declaration. See Section 3.3 [Rings and orderings], page 28 for a list of the available coefficient fields.

Example:

```
// finite field Z/p, p<= 32003
ring r = 32003,(x,y,z),dp;
number n = 4/6;
n;
```

```
↦ -10667
  // finite field GF(p^n), p^n <= 32767
  // z is a primitive root of the minimal polynomial
  ring rg= (7^2,z),x,dp;
  number n = 4/9+z;
  n;
↦ z11
  // the rational numbers
  ring r0 = 0,x,dp;
  number n = 4/6;
  n;
↦ 2/3
  // algebraic extensions of Z/p or Q
  ring ra=(0,a),x,dp;
  minpoly=a^2+1;
  number n=a3+a2+2a-1;
  n;
↦ (a-2)
  a^2;
↦ -1
  // transcedental extensions of Z/p or Q
  ring rt=(0,a),x,dp;
  number n=a3+a2+2a-1;
  n;
↦ (a3+a2+2a-1)
  a^2;
↦ (a2)
  // machine floating point numbers, single precision
  ring R_0=real,x,dp;
  number n=4/6;
  n;
↦ 6.667e-01
  n=0.25e+2;
  n;
↦ 2.500e+01
  // floating point numbers, arbitrary prescribed precision
  ring R_1=(real,50),x,dp;
  number n=4.0/6;
  n;
↦ 0.66666666666666666666666666666666666666666666666667
  n=0.25e+2;
  n;
↦ 25
  // floating point complex numbers, arbitrary prescribed precision
  // the third parameter gives the name of the imaginary unit
  ring R_2=(complex,50,i),x,dp;
  number n=4.0/6;
  n;
↦ 0.66666666666666666666666666666666666666666666666667
  n=0.25e+2*i+n;
  n;
↦ (0.66666666666666666666666666666666666666666666666667+i*25)
```

4.11.2 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 73)
2. a floating point number (if the coefficient field is `real`):
   <digits>.<digits>e<sign><digits>
3. an identifier of type number
4. a function returning number
5. an int expression (see Section 3.5.5 [Type conversion and casting], page 42)
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 3.5.5 [Type conversion and casting], page 42; Section 4.16 [ring], page 113.

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 if it is already a number, see Section 6.3 [Miscellaneous oddities], page 228.
For the behavior of comparison operators in rings with ground field different from real or the rational numbers, see Section 4.3.5 [boolean expressions], page 76.

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; // for int's exponent must be non-negative
↦    ? exponent must be non-negative
↦    ? error occurred in line 12: `  2^-2; // for int's exponent must be no\
   n-negative`
  number(2)^-2;
↦ 1/4
  3/4>=2/5;
↦ 1
  2/6==1/3;
↦ 1
```

4.11.4 number related functions

`cleardenom`
cancel denominators of numbers in poly and divide it by its content (see Section 5.1.7 [cleardenom], page 127)

`impart` imaginary part of a complex number, 0 otherwise (see Section 5.1.49 [impart], page 152, Section 5.1.108 [repart], page 193)

`numerator, denominator`
: return the numerator/denominator of a rational number (see Section D.2.4.15 [numerator], page 318, Section D.2.4.16 [denominator], page 318)

`leadcoef` coefficient of the leading term (see Section 5.1.62 [leadcoef], page 161)

`par` n-th parameter of the basering (see Section 5.1.92 [par], page 181)

`pardeg` degree of a number in ring parameters (see Section 5.1.93 [pardeg], page 182)

`parstr` string form of ring parameters (see Section 5.1.94 [parstr], page 182)

`repart` real part of a complex number (see Section 5.1.49 [impart], page 152, Section 5.1.108 [repart], page 193)

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 107). Polynomials can only be defined or accessed with respect to a basering which determines the coefficient type, the names of the indeterminates and the monomial ordering.

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

4.12.1 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;
f;
↦ x37y5z4+3x37+xy9
short = 0;
f;
↦ x^37*y^5*z^4+3*x^37+x*y^9
```

See also: Section 5.3.7 [short], page 222.

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. an int expression (see Section 3.5.5 [Type conversion and casting], page 42)
6. 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
  leadmonom(f);
↦ x2y3
  simplify(f,1);     // normalize leading coefficient
↦ x2y3+1/5x2y2-1/5xy-1/10x+1/10y+1/5
  poly g = 1/2x2 + 1/3y;
  cleardenom(g);
↦ 3x2+2y
  int i = 102;
  poly(i);
↦ 102
  typeof(_);
↦ poly
```

See Section 3.5.5 [Type conversion and casting], page 42; Section 4.16 [ring], page 113.

4.12.3 poly operations

`+` addition

`-` negation or subtraction

`*` multiplication

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

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

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

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

Example:

```
  ring R=0,(x,y),dp;
  poly f = x3y2 + 2x2y2 + xy - x + y + 1;
  f;
```

```
↦ x3y2+2x2y2+xy-x+y+1
  f + x5 + 2;
↦ x5+x3y2+2x2y2+xy-x+y+3
  f * x2;
↦ x5y2+2x4y2+x3y-x3+x2y+x2
  (x+y)/x;
↦ 1
  f/3x2;
↦ 1/3xy2+2/3y2
  x5 > f;
↦ 1
  x<=y;
↦ 0
  x>y;
↦ 1
  ring r=0,(x,y),ds;
  poly f = fetch(R,f);
  f;
↦ 1-x+y+xy+2x2y2+x3y2
  x5 > f;
↦ 0
  f[2..4];
↦ -x+y+xy
  size(f);
↦ 6
  f[size(f)+1]; f[-1];    // monomials out of range are 0
↦ 0
↦ 0
  intvec v = 6,1,3;
  f[v];          // the polynom built from the 1st, 3rd and 6th monomial of f
↦ 1+y+x3y2
```

4.12.4 poly related functions

`cleardenom`
: cancel denominators of numbers in poly and divide it by its content (see Section 5.1.7 [cleardenom], page 127)

`coef`
: matrix of coefficients and monomials (see Section 5.1.9 [coef], page 127)

`coeffs`
: matrix of coefficients (see Section 5.1.10 [coeffs], page 128)

`deg`
: degree (see Section 5.1.14 [deg], page 131)

`det`
: determinant (see Section 5.1.17 [det], page 133)

`diff`
: partial derivative (see Section 5.1.18 [diff], page 133)

`extgcd`
: Bezout representation of gcd (see Section 5.1.28 [extgcd], page 139)

`factorize`
: factorize polynomial (see Section 5.1.30 [factorize], page 140)

`finduni`
: find univariate polynomials in a zero-dimensional ideal (see Section 5.1.36 [finduni], page 143)

`gcd`
: greatest common divisor (see Section 5.1.39 [gcd], page 146)

homog homogenization (see Section 5.1.46 [homog], page 150)

jacob ideal, resp. matrix, of all partial derivatives (see Section 5.1.54 [jacob], page 155)

lead leading term (see Section 5.1.61 [lead], page 160)

leadcoef coefficient of the leading term (see Section 5.1.62 [leadcoef], page 161)

leadexp the exponent vector of the leading monomial (see Section 5.1.63 [leadexp], page 161)

leadmonom
leading monomial (see Section 5.1.64 [leadmonom], page 162)

jet monomials with degree smaller `k+1` (see Section 5.1.55 [jet], page 156)

ord degree of the leading monomial (see Section 5.1.90 [ord], page 180)

qhweight quasihomogeneous weights (see Section 5.1.100 [qhweight], page 187)

reduce normal form with respect to a standard base (see Section 5.1.106 [reduce], page 191)

rvar test for ring variable (see Section 5.1.112 [rvar], page 195)

simplify normalize a polynomial (see Section 5.1.114 [simplify], page 196)

size number of monomials (see Section 5.1.115 [size], page 197)

subst substitute a ring variable (see Section 5.1.123 [subst], page 205)

trace trace of a matrix (see Section 5.1.126 [trace], page 208)

var the indicated variable of the ring (see Section 5.1.132 [var], page 211)

varstr variable in string form (see Section 5.1.133 [varstr], page 211)

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 3.7 [Procedures], page 47. A file containing procedure definitions which comply with certain syntax rules is called a library. Such a file is loaded using the command `LIB`. For more information on libraries see Section 3.8 [Libraries], page 51.

4.13.1 proc declaration

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

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 3.7.3 [Parameter list], page 49. 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 5.1.65 [LIB], page 162 command.
Specifying `static` in front of the proc-definition is only possible in a library file and 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 5.1.65 [LIB], page 162; Section 3.8 [Libraries], page 51; Section 3.7 [Procedures], page 47.

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 3.3 [Rings and orderings], page 28.

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 computation of resolutions. It furthermore enables the use of partial results to compute, for example, Betti numbers or minimal resolutions. Like ideals and modules, a resolution can only be defined w.r.t. a basering.

Note: to access the elements of a resolution, it has to be assigned to a list, 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
↦ R <--  R <--  R
↦
↦ 0      1      2
↦ 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 3.5.5 [Type conversion and casting], page 42.

4.15.3 resolution related functions

`betti` Betti numbers of a resolution (see Section 5.1.3 [betti], page 124)

`lres` free resolution (see Section 5.1.69 [lres], page 165)

`minres` minimize a free resolution (see Section 5.1.74 [minres], page 168)

`mres` minimal free resolution of an ideal, resp. module w.r.t. a minimal set of generators of the given ideal, resp. module (see Section 5.1.78 [mres], page 171)

`res` free resolution of an ideal, resp. module, but not changing the given ideal, resp. module (see Section 5.1.109 [res], page 193)

`sres` free resolution of a standard basis (see Section 5.1.118 [sres], page 200)

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 3.3 [Rings and orderings], page 28.

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`
4. an expression_list of the name `real` and an int_expression.
5. an expression_list of the name `complex`, an optional int_expression and a name.

For the definition of the 'coefficient_field', see Section 3.3 [Rings and orderings], page 28.

'names_of_ring_variables' must be a list of names or indexed names.

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

1. `lp`, `dp`, `Dp`, `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 3.3.3 [Term orderings], page 32, Section B.2 [Monomial orderings], page 275.

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 coefficient field of a ring (see Section 5.1.6 [charstr], page 126)

`keepring` move ring to next upper level (see Section 5.2.8 [keepring], page 217)

`npars` number of ring parameters (see Section 5.1.84 [npars], page 175)

`nvars` number of ring variables (see Section 5.1.87 [nvars], page 176)

`ordstr` monomial ordering of a ring (see Section 5.1.91 [ordstr], page 181)

`parstr` names of all ring parameters or the name of the n-th ring parameter (see Section 5.1.94 [parstr], page 182)

`qring` quotient ring (see Section 4.14 [qring], page 111)

`setring` set a new basering (see Section 5.1.113 [setring], page 196)

`varstr` names of all ring variables or the name of the n-th ring variable (see Section 5.1.133 [varstr], page 211)

4.16.3 ring operations

`+` construct a new ring $k[X,Y]$ from $k_1[X]$ and $k_2[Y]$.

Concerning the ground fields k_1 and k_2 take the following guide lines into consideration:

- Neither k_1 nor k_2 may be R or C .
- If the characteristic of k_1 and k_2 differs, then one of them must be Q .
- At most one of k_1 and k_2 may be have parameters.
- If one of k_1 and k_2 is an algebraic extension of Z/p it may not be defined by a `charstr` of type `(p^n,a)`.

Example:

```
  ring R1=0,(x,y),dp;
  ring R2=32003,(a,b),dp;
  def R=R1+R2;
  R;
↦ //   characteristic : 32003
↦ //   number of vars : 4
↦ //        block   1 : ordering dp
↦ //                  : names    x y
↦ //        block   2 : ordering dp
↦ //                  : names    a b
↦ //        block   3 : ordering C
```

See also: Section D.2.6 [ring_lib], page 326.

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 5.1.26 [execute], page 138. They are also return values of certain interpreter related functions (see Section 5.1 [Functions], page 121). String constants consist of a sequence of ANY characters (including newline!) between a starting `"` and a closing `"`. There is also a string constant `newline`, which is the newline character. The `+` sign "adds" strings, `""` is the empty string (hence strings form a semigroup). Strings may be used to comment the output of a computation or to give it a nice format. Strings may also be used for intermediate conversion of one type into another.

```
  string s="Hi";
  string s1="a string with new line at the end"+newline;
  string s2="another string with new line at the end
  ";
  s;s1;s2;
↦ Hi
↦ a string with new line at the end
↦
↦ another string with new line at the end
↦
  ring r; ideal i=std(ideal(x,y^3));
  "dimension of i =",dim(i),", multiplicity of i =",mult(i);
↦ dimension of i = 1 , multiplicity of i = 3
  "dimension of i = "+string(dim(i))+", multiplicity of i = "+string(mult(i));
↦ dimension of i = 1, multiplicity of i = 3
  "a"+"b","c";
↦ ab c
```

A comma between two strings makes an expression list out of them (such a list is printed with a separating blank in between), while a `+` concatenates strings.

4.17.1 string declarations

Syntax: `string` name = string_expression ;
`string` name = list_of_string_expressions ;

Purpose: defines a string variable.

Default: `""` (the empty string)

Example:

```
  string s1="Now I know";
  string s2="how to encode a \" in a string...";
  string s=s1+" "+s2; // concatenation of 3 strings
  s;
↦ Now I know how to encode a " in a string...
  s1,s2;   // 2 strings, separated by a blank in the output:
↦ Now I know how to encode a " in a string...
```

4.17.2 string expressions

A string expression is:

1. a sequence of characters between two unescaped quotes (`"`)
2. an identifier of type string
3. a function returning string
4. a substring (using the bracket operator)
5. a type cast to string (see Section 4.17.3 [string type cast], page 116)
6. string expressions combined by the operation `+`.

Example:

```
// string_expression[start, length] : a substring
// (possibly filled up with blanks)
```

```
// the substring of s starting at position 2
// with a length of 4
string s="123456";
s[2,4];
↦ 2345
"abcd"[2,2];
↦ bc
// string_expression[position] : a character from a string
s[3];
↦ 3
// string_expression[position..position] :
// a substring starting at the first position up to the second
// given position
s[2..4];
↦ 2 3 4
// a function returning a string
typeof(s);
↦ string
```

See Section 3.5.5 [Type conversion and casting], page 42; Section 4.17.3 [string type cast], page 116.

4.17.3 string type cast

Syntax: `string` (expression [, expression_2, ... expression_n])

Type: string

Purpose: Converts each expression to a string, where expression can be of any type. The concatenated string of all concersions is returned.

The elements of intvec, intmat, ideal, module, matrix, and list, are separated by a comma. No newlines are inserted.
Not defined elements of a list are omitted.
For link, the name of the link is used.
For map, the ideal defining the mapping is converted.

Note: When applied to a list, elements of type intvec, intmat, ideal, module, matrix, and list become indistinguishable.

Example:

```
  string("1+1=", 2);
↦ 1+1=2
  string(intvec(1,2,3,4));
↦ 1,2,3,4
  string(intmat(intvec(1,2,3,4), 2, 2));
↦ 1,2,3,4
  ring r;
  string(r);
↦ (32003),(x,y,z),(dp(3),C)
  string(ideal(x,y));
↦ x,y
  qring R = std(ideal(x,y));
  string(R);
↦ (32003),(x,y,z),(dp(3),C)
  map phi = r, ideal(x,z);
```

```
  string(phi);
↦ x,z
  list l;
  string(l);
↦
  l[3] = 1;
  string(l); // notice that l[1],l[2] are omitted
↦ 1
  l[2] = l;
  l;
↦ [2]:
↦    [3]:
↦       1
↦ [3]:
↦    1
  string(l); // notice that lists of list is flattened
↦ 1,1
  l[1] = intvec(1,2,3);
  l;
↦ [1]:
↦    1,2,3
↦ [2]:
↦    [3]:
↦       1
↦ [3]:
↦    1
  string(l); // notice that intvec elements are not distinguishable
↦ 1,2,3,1,1
```

See Section 3.5.5 [Type conversion and casting], page 42; Section 5.1.97 [print], page 183; Section 4.17 [string], page 114.

4.17.4 string operations

`+`
: concatenation

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

string_expression `[` int_expression `]`
: is a character of the string; the index 1 gives the first character.

string_expression `[` int_expression`,` int_expression `]`
: is a substring, where the first argument is the start index and the second is the length of the substring, filled up with blanks if the length exceeds the total size of the string

string_expression `[` intvec_expression `]`
: is a expression list of characters from the string

Example:

```
  string s="abcde";
  s[2];
↦ b
  s[3,2];
```

```
↦ cd
  ">>"+s[1,10]+"<<";
↦ >>abcde     <<
  s[2]="BC"; s;
↦ aBcde
  intvec v=1,3,5;
  s=s[v]; s;
↦ ace
  s="123456"; s=s[3..5]; s;
↦ 345
```

4.17.5 string related functions

`charstr` description of the coefficient field of a ring (see Section 5.1.6 [charstr], page 126)

`execute` executing string as command (see Section 5.1.26 [execute], page 138)

`find` position of a substring in a string (see Section 5.1.35 [find], page 143)

`names` list of strings of all user-defined variable names (see Section 5.1.82 [names], page 173)

`nameof` name of an object (see Section 5.1.81 [nameof], page 173)

`option` lists all defined options (see Section 5.1.89 [option], page 177)

`ordstr` monomial ordering of a ring (see Section 5.1.91 [ordstr], page 181)

`parstr` names of all ring parameters or the name of the n-th ring parameter (see Section 5.1.94 [parstr], page 182)

`read` read a file (see Section 5.1.105 [read], page 190)

`size` length of a string (see Section 5.1.115 [size], page 197)

`sprintf` string formatting (see Section 5.1.117 [sprintf], page 199)

`typeof` type of an object (see Section 5.1.129 [typeof], page 209)

`varstr` names of all ring variables or the name of the n-th ring variable (see Section 5.1.133 [varstr], page 211)

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 non-zero coefficient. Since generators with zero coefficients need not be written any vector may be considered also as an element of a free module of higher rank. Like polynomials they can only be defined or accessed with respect to the 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 274).

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` ↦ `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 3.5.5 [Type conversion and casting], page 42; Section 4.16 [ring], page 113.

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 127)

`coeffs`
: matrix of coefficients (see Section 5.1.10 [coeffs], page 128)

`deg`
: degree (see Section 5.1.14 [deg], page 131)

`diff`
: partial derivative (see Section 5.1.18 [diff], page 133)

`gen`
: i-th generator (see Section 5.1.40 [gen], page 146)

`homog`
: homogenization (see Section 5.1.46 [homog], page 150)

`jet`
: k-jet: monomials with degree smaller `k+1` (see Section 5.1.55 [jet], page 156)

`lead`
: leading term (see Section 5.1.61 [lead], page 160)

`leadcoef`
: leading coefficient (see Section 5.1.62 [leadcoef], page 161)

`leadexp`
: the exponent vector of the leading monomial (see Section 5.1.63 [leadexp], page 161)

`leadmonom`
: leading monomial (see Section 5.1.64 [leadmonom], page 162)

`nrows`
: number of rows (see Section 5.1.86 [nrows], page 176)

`ord`
: degree of the leading monomial (see Section 5.1.90 [ord], page 180)

`reduce`
: normal form with respect to a standard base (see Section 5.1.106 [reduce], page 191)

`simplify`
: normalize a vector (see Section 5.1.114 [simplify], page 196)

`size`
: number of monomials (see Section 5.1.115 [size], page 197)

`subst`
: substitute a ring variable (see Section 5.1.123 [subst], page 205)

5 Functions and system variables

5.1 Functions

This section gives a complete reference of all functions, commands and special variables of the SINGULAR kernel (i.e., all built-in commands). See Section D.1 [standard_lib], page 293, for those functions from the `standard.lib` (this library is automatically loaded at start-up time) which extend the functionality of the kernel and are written in the SINGULAR programming language.

The general syntax of a function is

[target =] function_name (<arguments>);

If no target is specified, the result is printed. In some cases (e.g., `export`, `keepring`, `kill`, `setring`, `type`) the brackets are optional. For 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 variable name. If the attribute is not defined for this variable, `attrib` returns the empty string.

Example:

```
  ring r=0,(x,y,z),dp;
  ideal I=std(maxideal(2));
  attrib(I,"isSB");
↦ 1
  // 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 variable name to the value expression.

Example:

```
  ring r=0,(x,y,z),dp;
  ideal I=maxideal(2); // the attribute "isSB" is not set
  vdim(I);
```

```
↦ // ** 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. Some of these are used by the kernel of SINGULAR and referred to as reserved attributes. Non-reserved attributes may be used, however, in procedures and can considerably speed up computations.

Reserved attributes:
(not all are in use at the moment)

`isSB` the standard basis property is set by all commands computing a standard basis like `groebner`, `std`, `stdhilb` etc.; used by `lift`, `dim`, `degree`, `mult`, `hilb`, `vdim`, `kbase`

`isHomog` 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 5.1.86 [nrows], page 176)

`withSB` value of type ideal, resp. module, is std

`withHilb` value of type intvec is hilb(_,1) (see Section 5.1.45 [hilb], page 149)

`withRes` value of type list is a free resolution

`withDim` value of type int is the dimension (see Section 5.1.19 [dim], page 134)

`withMult` value of type int is the multiplicity (see Section 5.1.80 [mult], page 172)

5.1.2 bareiss

Syntax: `bareiss (` module_expression `)`
`bareiss (` matrix_expression `)`
`bareiss (` module_expression`,` int_expression`,` int_expression `)`
`bareiss (` matrix_expression`,` int_expression`,` int_expression `)`

Type: list of module and intvec

Purpose: applies the sparse Gauss-Bareiss algorithm (see Section C.7 [References], page 291, Lee and Saunders) to a module (or with type conversion to a matrix) with an 'optimal' pivot strategy. The vectors of the module are the columns of the matrix, hence elimination takes place w.r.t. rows.
With only one parameter a complete elimination is done. Result is a list: the first entry is a module with a minimal independent set of vectors (as a matrix lower triangular), the second entry an intvec with the permutation of the rows w.r.t. the original matrix, that is, a k at position l indicates that row l became row k.
The further parameters control the algorithm. `bareiss(M,i,j)` does not consider the last i rows in the elimination procedure and stops computing when the remaining number of vectors (columns) to reduce is at most j.

Example:

```
  ring r=0,(x,y,z),(c,dp);
  module mm;
  // ** generation of the module mm **
  int d=7;
  int b=2;
  int db=d-b;
  int i;
  for(i=d;i>0;i--){ mm[i]=3*x*gen(i); }
  for(i=db;i;i--){ mm[i]=mm[i]+7*y*gen(i+b); }
  for(i=d;i>db;i--){ mm[i]=mm[i]+7*y*gen(i-db); }
  for(i=d;i>b;i--){ mm[i]=mm[i]+11*z*gen(i-b); }
  for(i=b;i;i--){ mm[i]=mm[i]+11*z*gen(i+db); }
  // ** the generating matrix of mm **
  print(mm);
↦ 3x, 0,  11z,0,  0,  7y, 0,
↦ 0,  3x, 0,  11z,0,  0,  7y,
↦ 7y, 0,  3x, 0,  11z,0,  0,
↦ 0,  7y, 0,  3x, 0,  11z,0,
↦ 0,  0,  7y, 0,  3x, 0,  11z,
↦ 11z,0,  0,  7y, 0,  3x, 0,
↦ 0,  11z,0,  0,  7y, 0,  3x
  // complete elimination
  list ss=bareiss(mm);
  print(ss[1]);
↦ 7y, 0,     0,     0,      0,        0,     0,
↦ 3x, -33xz, 0,     0,      0,        0,     0,
↦ 11z,-121z2,1331z3,0,      0,        0,     0,
↦ 0,  0,     0,     9317yz3,0,        0,     0,
↦ 0,  21xy,  _[5,3],14641z4,-43923xz4,0,     0,
↦ 0,  0,     0,     0,      65219y2z3,_[6,6],0,
↦ 0,  49y2,  _[7,3],3993xz3,_[7,5],   _[7,6],_[7,7]
  ss[2];
↦ 2,7,5,1,4,3,6
  // elimination up to 3 vectors
  ss=bareiss(mm,0,3);
  print(ss[1]);
↦ 7y, 0,     0,     0,      0,        0,        0,
↦ 3x, -33xz, 0,     0,      0,        0,        0,
↦ 11z,-121z2,1331z3,0,      0,        0,        0,
↦ 0,  0,     0,     9317yz3,0,        0,        0,
↦ 0,  0,     0,     0,      27951xyz3,102487yz4,65219y2z3,
↦ 0,  21xy,  _[6,3],14641z4,_[6,5],   _[6,6],   -43923xz4,
↦ 0,  49y2,  _[7,3],3993xz3,_[7,5],   _[7,6],   _[7,7]
  ss[2];
↦ 2,7,5,1,3,4,6
  // elimination without the last 3 rows
  ss=bareiss(mm,3,0);
  print(ss[1]);
↦ 7y, 0,   0,      0,       0,     0,     0,
↦ 0,  77yz,0,      0,       0,     0,     0,
↦ 0,  0,   231xyz, 0,       0,     0,     0,
↦ 0,  0,   0,      1617xy2z,0,     0,     0,
↦ 11z,21xy,-1331z3,14641z4, _[5,5],_[5,6],_[5,7],
```

```
↦ 0,  0,   539y2z, _[6,4],  _[6,5],_[6,6],-3773y3z,
↦ 3x, 49y2,-363xz2,3993xz3, _[7,5],_[7,6],_[7,7]
  ss[2];
↦ 2,3,4,1
```

See Section 5.1.17 [det], page 133; Section 4.9 [matrix], page 96.

5.1.3 betti

Syntax: `betti` (list_expression)
`betti` (resolution_expression)
`betti` (list_expression , int_expression)
`betti` (resolution_expression , int_expression)

Type: intmat

Purpose: with 1 argument: computes the graded Betti numbers of a minimal resolution of R^n/M, if R denotes the basering and M a homogeneous submodule of R^n and the argument represents a resolution of R^n/M.
The entry d of the intmat at place (i,j) is the minimal number of generators in degree i+j of the j-th syzygy module (= module of relations) of R^n/M (the 0th (resp. 1st) syzygy module of R^n/M is R^n (resp. M)). The argument is considered to be the result of a res/sres/mres/nres/lres command. This implies that a zero is only allowed (and counted) as a generator in the first module.
For the computation betti uses only the initial monomials. This could lead to confusing results for a non-homogeneous input.

The optional second argument is a switch for the minimization of the Betti numbers. If it is 0 then the Betti numbers correspond exactly to the input - otherwise the command is identical to the one-argument form.

Example:

```
  ring r=32003,(a,b,c,d),dp;
  ideal j=bc-ad,b3-a2c,c3-bd2,ac2-b2d;
  list T=mres(j,0); // 0 forces a full resolution
  // a minimal set of generators for j:
  print(T[1]);
↦ bc-ad,
↦ c3-bd2,
↦ ac2-b2d,
↦ b3-a2c
  // second syzygy module of r/j which is the first
  // syzygy module of j (minimal generating set):
  print(T[2]);
↦ bd,c2,ac,b2,
↦ -a,-b,0, 0,
↦ c, d, -b,-a,
↦ 0, 0, -d,-c
  // the second syzygy module (minimal generating set):
  print(T[3]);
↦ -b,
↦ a,
↦ -c,
↦ d
  print(T[4]);
```

```
↦ 0
  betti(T);
↦ 1,0,0,0,
↦ 0,1,0,0,
↦ 0,3,4,1
  // most useful for reading off the graded Betti numbers:
  print(betti(T),"betti");
↦            0     1     2     3
↦ ------------------------------
↦     0:     1     -     -     -
↦     1:     -     1     -     -
↦     2:     -     3     4     1
↦ ------------------------------
↦ total:     1     4     4     1
```

Hence

- the 0th syzygy module of r/j (which is r) has 1 generator in degree 0 (which is 1),
- the 1st syzygy module `T[1]` (which is j) has 4 generators (one in degree 2 and three in degree 3),
- the 2nd syzygy module `T[2]` has 4 generators (all in degree 4),
- the 3rd syzygy module `T[3]` has 1 generator in degree 5,

where the generators are the columns of the displayed matrix and degrees are assigned such that the corresponding maps have degree 0:

$$0 \longleftarrow r/j \longleftarrow r(1) \xleftarrow{T[1]} r(2) \oplus r^3(3) \xleftarrow{T[2]} r^4(4) \xleftarrow{T[3]} r(5) \longleftarrow 0 \quad .$$

See Section C.3 [Syzygies and resolutions], page 282; Section 5.1.47 [hres], page 151; Section 5.1.69 [lres], page 165; Section 5.1.78 [mres], page 171; Section 5.1.97 [print], page 183; Section 5.1.109 [res], page 193; Section 4.15 [resolution], page 112; Section 5.1.118 [sres], page 200.

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 5.1.6 [charstr], page 126; Section 4.16 [ring], page 113.

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 ordering of variables.
One application is the decomposition of the zero-set.

Example:

```
  ring r=32003,(x,y,z),dp;
  print(char_series(ideal(xyz,xz,y)));
↦ y,z,
↦ x,y
```

See Section C.4 [Characteristic sets], page 283.

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 5.1.4 [char], page 125; Section 5.1.91 [ordstr], page 181; Section 4.16 [ring], page 113; Section 5.1.133 [varstr], page 211.

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.

Example:

```
  ring r=0,(x,y,z),dp;
  poly f=(3x+6y)^5;
  f/5;
↦ 243/5x5+486x4y+1944x3y2+3888x2y3+3888xy4+7776/5y5
  cleardenom(f/5);
↦ x5+10x4y+40x3y2+80x2y3+80xy4+32y5
```

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 83; Section 5.1.88 [open], page 177.

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 monomials 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 second matrix (i.e., the 4th argument) contains the monomials, the first matrix (i.e., the 3rd argument) the corresponding coefficients of the monomials in the vector.

Note: coef considers only monomials which really occur in f (i.e., which are not 0), while coeffs (see Section 5.1.10 [coeffs], page 128) returns the coefficient 0 at the appropriate place if a monomial is not present.

Example:

```
  ring r=32003,(x,y,z),dp;
  poly f=x5+5x4y+10x2y3+y5;
  matrix m=coef(f,y);
  print(m);
↦ y5,y3,  y,  1,
↦ 1, 10x2,5x4,x5
  f=x20+xyz+xy+x2y+z3;
  print(coef(f,xy));
↦ x20,x2y,xy, 1,
↦ 1,  1,  z+1,z3
  vector v=[f,zy+77+xy];
  print(v);
↦ [x20+x2y+xyz+z3+xy,xy+yz+77]
  matrix mc; matrix mm;
  coef(v,y,mc,mm);
  print(mc);
↦ x2+xz+x,x20+z3,
↦ x+z,    77
  print(mm);
↦ y,1,
↦ y,1
```

See Section 5.1.10 [coeffs], page 128.

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: develops each polynomial of the first argument, say J, 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 occurring polynomials
k = 1 if J is a polynomial,
k = number of generators if J is an ideal.

If J is a vector or a module this procedure is repeated for each component and the resulting matrices are appended.
The third argument is used to return the matrix T of coefficients such that `matrix`(J) = T*M.

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.
If $M = (m_{ij})$ then 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

$$J_{i,j} = z^0 \cdot m_{(i-1)d+1,j} + z^1 \cdot m_{(i-1)d+2,j} + ... + z^{d-1} \cdot m_{id,j}.$$

Example:

```
  ring r;
  poly f=(x+y)^3;
  matrix M=coeffs(f,y);
  print(M);
↦ x3,
↦ 3x2,
↦ 3x,
↦ 1
  ideal i=f,xyz+z10y4;
  print(coeffs(i,y));
↦ x3, 0,
↦ 3x2,xz,
↦ 3x, 0,
↦ 1,  0,
↦ 0,  z10
```

Syntax: `coeffs` (ideal_expression, ideal_expression)
`coeffs` (module_expression, module_expression)
`coeffs` (ideal_expression, ideal_expression, product_of_ringvars)
`coeffs` (module_expression, module_expression, product_of_ringvars)

Type: matrix

Purpose: let the first argument be M, the second argument be K (a set of monomials, resp. vectors with monomial entries, in the variables appearing in P), the third argument be the product P of variables to consider (if this argument is not given, then the product of all ring variables is taken as default argument).
M is supposed to consist of elements of (resp. have entries in) a finitely generated module over a ring in the variables not appearing in P. K should contain the generators of M over this smaller ring. 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
  print(coeffs(M,ideal(x2,xy,y2)));
↦ 0,0,
↦ 0,1,
↦ 0,0
```

See Section 5.1.9 [coef], page 127; Section 5.1.56 [kbase], page 157.

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:

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

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

Example:

```
  ring r=0,(a,b,c,d),dp;
  ideal I=a2,a2+bc,abc;
  ideal J=a2-bc,abcd;
  print(contract(I,J));
↦ 1,0,
↦ 0,ad,
↦ 0,d
```

See Section 5.1.18 [diff], page 133.

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`.

Note: See Section 3.9 [Guidelines for writing a library], page 53, for an example how this is used for displaying comments while procedures are executed.

Example:

```
  int debug=0;
  intvec i=1,2,3;
  dbprint(debug,i);
  debug=1;
  dbprint(debug,i);
↦ 1,2,3
  voice;
↦ 1
  printlevel;
↦ 0
  dbprint(i);
```

See Section 3.10 [Debugging tools], page 63; Section 5.1.97 [print], page 183; Section 5.3.6 [printlevel], page 221; Section 5.3.11 [voice], page 224.

5.1.13 defined

Syntax: `defined` (name)

Type: int

Purpose: returns a value `<>0` (TRUE) if there is a user-defined object with this name, and 0 (FALSE) otherwise.
A non-zero return value is the level where the object is defined (level 1 denotes the top level, level 2 the level of a first procedure, level 3 the level of a procedure called by a first procedure, etc.). For ring variables, -1 is returned.

Note: A local object `m` may be identified by `if (defined(m)==voice)`.

Example:

```
  ring r=(0,t),(x,y),dp;
  matrix m[5][6]=x,y,1,2,0,x+y;
  defined(mm);
↦ 0
  defined(r) and defined(m);
↦ 1
  defined(m)==voice;   // m is defined in the current level
↦ 1
  defined(x);
↦ -1
  defined(z);
↦ 0
  defined(t);
↦ -1
  defined(42);
↦ -1
```

See Section 5.1.112 [rvar], page 195; Section 5.3.11 [voice], page 224.

5.1.14 deg

Syntax: `deg` (poly_expression)
`deg` (vector_expression)
`deg` (poly_expression `,` intvec_expression)
`deg` (vector_expression `,` intvec_expression)

Type: int

Purpose: returns the maximal (weighted) degree of the terms of a polynomial or a vector;
deg(0) is -1.
The optional second argument gives the weight vector, the default weights of the base ring will be used otherwise.

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 5.1.55 [jet], page 156; Section 5.1.90 [ord], page 180; Section 4.12 [poly], page 107; Section 4.18 [vector], page 118.

5.1.15 degree

Syntax: `degree` (ideal_expression)
`degree` (module_expression)

Type: int

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

Example:

```
ring r3=32003,(x,y,z,h),dp;
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);
i=homog(i,h);
ideal i0=std(i);
degree(i0);
↦ 720
```

See Section 5.1.19 [dim], page 134; Section 4.2 [ideal], page 68; Section 5.1.80 [mult], page 172; Section 5.1.120 [std], page 202; Section 5.1.134 [vdim], page 212.

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 5.1.51 [insert], page 154; Section 4.7 [list], page 91.

5.1.17 det

Syntax: `det` (intmat_expression)
`det` (matrix_expression)
`det` (module_expression)

Type: int, resp. poly

Purpose: returns the determinant of a square matrix. A module is considered as a matrix. The applied algorithms depend on type of input. If the input is a module or matrix with symbolic entries the Bareiss algorithm is used. In the other cases the chinese remainder algorithm is used. For large sparse problems the input as a module has advantages.

Example:

```
  ring r=7,(x,y),wp(2,3);
  matrix m[3][3]=1,2,3,4,5,6,7,8,x;
  det(m);
↦ -3x-1
```

See Section 4.4 [intmat], page 78; Section 4.9 [matrix], page 96; Section 5.1.73 [minor], page 168.

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 an m x n matrix.

Example:

```
  ring r=0,(x,y,z),dp;
  poly f=2x3y+3z5;
  diff(f,x);
↦ 6x2y
  vector v=[f,y2+z];
  diff(v,z);
```

```
↦ 15z4*gen(1)+gen(2)
  ideal j=x2-yz,xyz;
  ideal i=x2,x2+yz,xyz;
  // corresponds to differential operators
  // d2/dx2, d2/dx2+d2/dydz, d3/dxdydz:
  print(diff(i,j));
↦ 2,0,
↦ 1,x,
↦ 0,1
```

See Section 5.1.11 [contract], page 130; Section 4.2 [ideal], page 68; Section 5.1.54 [jacob], page 155; Section 4.9 [matrix], page 96; Section 4.10 [module], page 100; Section 4.12 [poly], page 107; Section 5.1.132 [var], page 211; Section 4.18 [vector], page 118.

5.1.19 dim

Syntax: `dim` (ideal_expression)
`dim` (module_expression)

Type: int

Purpose: computes the dimension of the ideal, resp. module, generated by the leading monomials of the given generators of the ideal, resp. module. This is also the dimension of the ideal if it is represented by a standard basis.

Note: The dimension of an ideal I means the Krull dimension of the basering modulo I.
The dimension of a module is the dimension of its annihilator ideal.

Example:

```
  ring r=32003,(x,y,z),dp;
  ideal I=x2-y,x3;
  dim(std(I));
↦ 1
```

See Section 5.1.15 [degree], page 132; Section 4.2 [ideal], page 68; Section 5.1.80 [mult], page 172; Section 5.1.120 [std], page 202; Section 5.1.134 [vdim], page 212.

5.1.20 division

Syntax: `division` (ideal_expression, ideal_expression)
`division` (module_expression, module_expression)
`division` (ideal_expression, ideal_expression, int_expression)
`division` (module_expression, module_expression, int_expression)
`division` (ideal_expression, ideal_expression, int_expression, intvec_expression)
`division` (module_expression, module_expression, int_expression, intvec_expression)

Type: list

Purpose: `division` computes a division with remainder. For two ideals resp. modules `M` (first argument) and `N` (second argument), it returns a list `T,R,U` where `T` is a matrix, `R` is an ideal resp. a module, and `U` is a diagonal matrix of units such that `matrix(M)*U=matrix(N)*T+matrix(R)` is a standard representation for the normal form `R` of `M` with respect to a standard basis of `N`. `division` uses different algorithms depending on whether `N` is represented by a standard basis. For a polynomial basering, the matrix `U` is the identity matrix. A matrix `T` as above is also computed by `lift`. For additional arguments `n` (third argument) and `w` (fourth argument), `division` returns a list `T,R` as above such that `matrix(M)=matrix(N)*T+matrix(R)` is a standard representation for the normal form `R` of `M` with respect to `N` up to weighted degree `n` with respect to the weight vector `w`. The weighted degree of `T` and `R` respect to `w` is at most `n`. If the weight vector `w` is not given, `division` uses the standard weight vector `w=1,...,1`.

Example:

```
ring R=0,(x,y),ds;
poly f=x5+x2y2+y5;
division(f,jacob(f));
↦ [1]:
↦    _[1,1]=1/5x
↦    _[2,1]=3/10y
↦ [2]:
↦    _[1]=-1/2y5
↦ [3]:
↦    _[1,1]=1
division(f^2,jacob(f));
↦ [1]:
↦    _[1,1]=1/20x6-9/80xy5-5/16x7y+5/8x2y6
↦    _[2,1]=1/8x2y3+1/5x5y+1/20y6-3/4x3y4-5/4x6y2-5/16xy7
↦ [2]:
↦    _[1]=0
↦ [3]:
↦    _[1,1]=1/4-25/16xy
division(ideal(f^2),jacob(f),10);
↦ // ** _ is no standardbasis
↦ [1]:
↦    _[1,1]=-75/8y9
↦    _[2,1]=1/2x2y3+x5y-1/4y6-3/2x3y4+15/4xy7+375/16x2y8
↦ [2]:
↦    _[1]=x10+9/4y10
```

See Section 4.2 [ideal], page 68; Section 5.1.66 [lift], page 163; Section 4.10 [module], page 100.

5.1.21 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 retrieve it later on.

Example:

```
ring r;
// write the whole session to the file dump.ascii
// in ASCII format
dump(":w dump.ascii");
kill r;                  // kill the basering
// reread the session from the file
```

```
  // redefining everything which was not explicitly killed before
  getdump("dump.ascii");
↦ // ** redefining stdfglm **
↦ // ** redefining stdhilb **
↦ // ** redefining groebner **
↦ // ** redefining res **
↦ // ** redefining quot **
↦ // ** redefining quot1 **
↦ // ** redefining quotient0 **
↦ // ** redefining quotient1 **
↦ // ** redefining quotient2 **
↦ // ** redefining quotient3 **
↦ // ** redefining quotient5 **
↦ // ** redefining quotient4 **
↦ // ** redefining intersect1 **
↦ // ** redefining sprintf **
↦ // ** redefining printf **
↦ // ** redefining fprintf **
  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 lists are dumped as one flatted list. Furthermore, links themselves are not dumped.

See Section 5.1.41 [getdump], page 146; Section 4.6 [link], page 83; Section 5.1.137 [write], page 213.

5.1.22 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 intersecting it with the subring not containing these variables.
`eliminate` does not need a special ordering nor a standard basis as input.

Note: Since elimination is expensive, for homogeneous input it might be useful first to compute the Hilbert function of the ideal (first argument) with a fast ordering (e.g., `dp`). Then make use of it to speed up the computation: a Hilbert-driven elimination uses the intvec provided as the third argument.

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 5.1.45 [hilb], page 149; Section 4.2 [ideal], page 68; Section 4.10 [module], page 100; Section 5.1.120 [std], page 202.

5.1.23 eval

Syntax: `eval` (expression)

Type: none

Purpose: evaluates (quoted) expressions. Within a quoted expression, the quote can be "undone" by an `eval` (i.e., each eval "undoes" the effect of exactly one quote). Used only when receiving a quoted expression from an 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 86; Section 5.1.102 [quote], page 188; Section 5.1.137 [write], page 213.

5.1.24 ERROR

Syntax: `ERROR` (string_expression)

Type: none

Purpose: Immediately interrupts the current computation, returns to the top-level, and displays the argument `string_expression` as error message.

Note: This should be used as an emergency, resp. failure, exit within procedures.

Example:

```
int i=1;
proc myError() {ERROR("Need to leave now");i=2;}
myError();
↦    ? Need to leave now
↦    ? error occurred in line 2: `ERROR("Need to leave now");i=2;`
↦    ? leaving ::myError
↦    skipping text from `;` error at token `)`
i;
↦ 1
```

5.1.25 example

Syntax: `example` topic ;

Purpose: computes an example for `topic`. Examples are available for all SINGULAR kernel and library functions. Where available (e.g., within Emacs), use `<TAB>` completion for a list of all available example `topics`.

Example:

```
example prime;
example intvec_declarations;
```

See also: Section 5.1.43 [help], page 148.

5.1.26 execute

Syntax: `execute` (string_expression)

Type: none

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

Note: The command `return` cannot appear in the string.
`execute` should be avoided in procedures whenever possible, since it may give rise to name conflicts. Moreover, such procedures cannot be precompiled (a feature which SINGULAR will provide in the future).

Example:

```
  ring r=32003,(x,y,z),dp;
  ideal i=x+y,z3+22y;
  write(":w save_i",i);
  ring r0=0,(x,y,z),Dp;
  string s="ideal k="+read("save_i")+";";
  s;
↦ ideal k=x+y,z3+22y
↦ ;
  execute(s); // define the ideal k
  k;
↦ k[1]=x+y
↦ k[2]=z3+22y
```

5.1.27 exit

Syntax: `exit;`

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

5.1.28 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 5.1.39 [gcd], page 146; Section 4.3 [int], page 72; Section 4.12 [poly], page 107.

5.1.29 facstd

Syntax: `facstd` (ideal_expression)
`facstd` (ideal_expression, ideal_expression)

Type: list of ideals

Purpose: returns a list of ideals computed by the factorizing Groebner basis algorithm.
The intersection of these ideals has the same zero-set as the input, i.e., the radical of the intersection coincides with the radical of the input ideal. In many (but not all!) cases this is already a decomposition of the radical of the ideal. (Note however, that, in general, no inclusion between the input and output ideals holds.)
The second, optional argument gives a list of polynomials which define non-zero constraints. 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 baserings over real ground fields, galois fields and over algebraic extensions over the rational numbers (that is, only implemented for ground fields for which Section 5.1.30 [factorize], page 140 is implemented).

Example:

```
  ring r=32003,(x,y,z),(c,dp);
  ideal I=xyz,x2z;
```

```
  facstd(I);
↦ [1]:
↦    _[1]=z
↦ [2]:
↦    _[1]=x
  facstd(I,x);
↦ [1]:
↦    _[1]=z
```

See Section 4.2 [ideal], page 68; Section 4.16 [ring], page 113; Section 5.1.120 [std], page 202.

5.1.30 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 and finite fields of type `(p^n,a)`.

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 107.

5.1.31 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 rings. `fetch` is the identity map between rings and qrings, the i-th variable of the source ring is mapped to the i-th variable of the basering. The coefficient fields must be compatible. (See Section 4.8 [map], page 94 for a description of possible mapping between different ground fields).
`fetch` offers a convenient way to change variable names or orderings, or to map objects from a ring to a quotient ring of that ring or vice versa.

Note: Compared with `imap`, `fetch` uses the position of the ring variables, not their names.

Example:

```
  ring r=0,(x,y,z),dp;
  ideal i=maxideal(2);
  ideal j=std(i);
  poly f=x+y2+z3;
  vector v=[f,1];
  qring q=j;
  poly f=fetch(r,f);
  f;
↦ z3+y2+x
  vector v=fetch(r,v);
  v;
↦ z3*gen(1)+y2*gen(1)+x*gen(1)+gen(2)
  ideal i=fetch(r,i);
  i;
↦ i[1]=z2
↦ i[2]=yz
↦ i[3]=y2
↦ i[4]=xz
↦ i[5]=xy
↦ i[6]=x2
  ring rr=0,(a,b,c),lp;
  poly f=fetch(q,f);
  f;
↦ a+b2+c3
  vector v=fetch(r,v);
  v;
↦ a*gen(1)+b2*gen(1)+c3*gen(1)+gen(2)
  ideal k=fetch(q,i);
  k;
↦ k[1]=c2
↦ k[2]=bc
↦ k[3]=b2
↦ k[4]=ac
↦ k[5]=ab
↦ k[6]=a2
```

See Section 5.1.48 [imap], page 152; Section 4.8 [map], page 94; Section 4.14 [qring], page 111; Section 4.16 [ring], page 113.

5.1.32 fglm

Syntax: `fglm` (ring_name, ideal_name)

Type: ideal

Purpose: computes for the given ideal in the given ring a reduced Groebner basis in the current ring, by applying the so-called FGLM (Faugere, Gianni, Lazard, Mora) algorithm.
The main application is to compute a lexicographical Groebner basis from a reduced Groebner basis with respect to a degree ordering. This can be much faster than computing a lexicographical Groebner basis directly.

Note: The ideal must be zero-dimensional and given as a reduced Groebner basis in the given ring.
The only permissible differences between the given ring and the current ring are the monomial ordering and a permutation of the variables, resp. parameters.

Example:

```
  ring r=0,(x,y,z),dp;
  ideal i=y3+x2, x2y+x2, x3-x2, z4-x2-y;
  option(redSB);   // force the computation of a reduced SB
  i=std(i);
  vdim(i);
↦ 28
  ring s=0,(z,x,y),lp;
  ideal j=fglm(r,i);
  j;
↦ j[1]=y4+y3
↦ j[2]=xy3-y3
↦ j[3]=x2+y3
↦ j[4]=z4+y3-y
```

See Section 5.1.33 [fglmquot], page 142; Section 5.1.89 [option], page 177; Section 4.14 [qring], page 111; Section 4.16 [ring], page 113; Section 5.1.120 [std], page 202; Section 5.1.121 [stdfglm], page 204; Section 5.1.134 [vdim], page 212.

5.1.33 fglmquot

Syntax: `fglmquot` (ideal_expression, poly_expression)

Type: ideal

Purpose: computes a reduced Groebner basis of the ideal quotient `I:p` of a zero-dimensional ideal `I` and a polynomial `p` using FGLM-techniques.

Note: The ideal must be zero-dimensional and given as a reduced Groebner basis in the given ring. The poly must be reduced with respect to the ideal.

Example:

```
  ring r=0,(x,y,z),lp;
  ideal i=y3+x2,x2y+x2,x3-x2,z4-x2-y;
  option(redSB);   // force the computation of a reduced SB
  i=std(i);
  poly p=reduce(x+yz2+z10,i);
  ideal j=fglmquot(i,p);
  j;
```

```
↦ j[1]=z12
↦ j[2]=yz4-z8
↦ j[3]=y2+y-z8-z4
↦ j[4]=x+y-z10-z6-z4
```

See Section 5.1.32 [fglm], page 142; Section 5.1.89 [option], page 177; Section 5.1.103 [quotient], page 189; Section 4.16 [ring], page 113; Section 5.1.120 [std], page 202; Section 5.1.134 [vdim], page 212.

5.1.34 files, input from

Syntax: `<` "filename"

Type: none

Purpose: input comes from the file filename. Shorthand for `execute(read(filename))`.

Example:

```
< "example"; //read in the file example and execute it
```

See Section 5.1.26 [execute], page 138; Section 5.1.105 [read], page 190.

5.1.35 find

Syntax: `find` (string_expression, substring_expression)
`find` (string_expression, substring_expression, int_expression)

Type: int

Purpose: returns the first position of the substring in the string or 0 (if not found),
starts the search at the position given in the 3rd argument.

Example:

```
  find("Aac","a");
↦ 2
  find("abab","a"+"b");
↦ 1
  find("abab","a"+"b",2);
↦ 3
  find("abab","ab",3);
↦ 3
  find("0123","abcd");
↦ 0
```

See Section 4.17 [string], page 114.

5.1.36 finduni

Syntax: `finduni` (ideal_expression)

Type: ideal

Purpose: returns an ideal which is contained in the ideal_expression such that the i-th generator is a univariate polynomial in the i-th ring variable.
The polynomials have minimal degree w.r.t. this property.

Note: The ideal must be zero-dimensional and given as a reduced Groebner basis in the current ring.

Example:

```
ring  r=0,(x,y,z), dp;
ideal i=y3+x2,x2y+x2,z4-x2-y;
option(redSB);  // force computation of reduced basis
i=std(i);
ideal k=finduni(i);
print(k);
↦ x4-x2,
↦ y4+y3,
↦ z12
```

See Section 5.1.89 [option], page 177; Section 4.16 [ring], page 113; Section 5.1.120 [std], page 202; Section 5.1.134 [vdim], page 212.

5.1.37 fprintf

Procedure from library `standard.lib` (see Section D.1 [standard_lib], page 293).

Syntax: `fprintf (` link_expression`,` string_expression `[,` any_expressions`] )`

Return: none

Purpose: `fprintf(l,fmt,...);` performs output formatting. The second argument is a format control string. Additional arguments may be required, depending on the content of the control string. A series of output characters is generated as directed by the control string; these characters are written to the link l. The control string `fmt` is simply text to be copied, except that the string may contain conversion specifications.
Do `help print;` for a listing of valid conversion specifications. As an addition to the conversions of `print`, the `%n` and `%2` conversion specification does not consume an additional argument, but simply generates a newline character.

Note: If one of the additional arguments is a list, then it should be enclosed once more into a `list()` command, since passing a list as an argument flattens the list by one level.

Example:

```
  ring r=0,(x,y,z),dp;
module m=[1,y],[0,x+z];
intmat M=betti(mres(m,0));
list l=r,m,M;
link li="";   // link to stdout
fprintf(li,"s:%s,l:%l",1,2);
↦ s:1,l:int(2)
fprintf(li,"s:%s",l);
↦ s:(0),(x,y,z),(dp(3),C)
fprintf(li,"s:%s",list(l));
↦ s:(0),(x,y,z),(dp(3),C),y*gen(2)+gen(1),x*gen(2)+z*gen(2),1,1
fprintf(li,"2l:%2l",list(l));
↦ 2l:list("(0),(x,y,z),(dp(3),C)",
↦ module(y*gen(2)+gen(1),
↦ x*gen(2)+z*gen(2)),
↦ intmat(intvec(1,1 ),1,2))
↦
fprintf(li,"%p",list(l));
↦ [1]:
↦    //   characteristic : 0
```

```
↦ //   number of vars : 3
↦ //        block   1 : ordering dp
↦ //                  : names    x y z
↦ //        block   2 : ordering C
↦ [2]:
↦    _[1]=y*gen(2)+gen(1)
↦    _[2]=x*gen(2)+z*gen(2)
↦ [3]:
↦    1,1
↦
fprintf(li,"%;",list(l));
↦ [1]:
↦    //   characteristic : 0
↦ //   number of vars : 3
↦ //        block   1 : ordering dp
↦ //                  : names    x y z
↦ //        block   2 : ordering C
↦ [2]:
↦    _[1]=y*gen(2)+gen(1)
↦    _[2]=x*gen(2)+z*gen(2)
↦ [3]:
↦    1,1
↦
fprintf(li,"%b",M);
↦            0     1
↦ ------------------
↦     0:     1     1
↦ ------------------
↦ total:     1     1
↦
```

See also: Section 5.1.97 [print], page 183; Section 5.1.98 [printf], page 185; Section 5.1.117 [sprintf], page 199; Section 4.17 [string], page 114.

5.1.38 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 5.1.40 [gen], page 146; Section 4.10 [module], page 100.

5.1.39 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 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 5.1.28 [extgcd], page 139; Section 4.3 [int], page 72; Section 4.12 [poly], page 107.

5.1.40 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 5.1.38 [freemodule], page 145; Section 4.3 [int], page 72; Section 4.18 [vector], page 118.

5.1.41 getdump

Syntax: `getdump` (link_expression)

Type: none

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

Example:

```
  int i=3;
  dump(":w example.txt");
  kill i;
```

```
option(noredefine);
getdump("example.txt");
i;
↦ 3
```

Restrictions:
getdump is not supported for DBM links, or for a link connecting to stdin (standard input).

See Section 5.1.21 [dump], page 135; Section 4.6 [link], page 83; Section 5.1.105 [read], page 190.

5.1.42 groebner

Procedure from library standard.lib (see Section D.1 [standard_lib], page 293).

Syntax: groebner (ideal_expression)
groebner (module_expression)
groebner (ideal_expression, int_expression)
groebner (module_expression, int_expression)

Type: type of the first argument

Purpose: computes the standard basis of the first argument I (ideal or module), by a heuristically chosen method: if the ordering of the current ring is a local ordering, or if it is a non-block ordering and the current ring has no parameters, then std(I) is returned. Otherwise, I is mapped into a ring with no parameters and ordering dp, where its Hilbert series is computed. This is followed by a Hilbert-series based std computation in the original ring.

Note: If a 2nd argument wait is given, then the computation proceeds at most wait seconds. That is, if no result could be computed in wait seconds, then the computation is interrupted, 0 is returned, a warning message is displayed, and the global variable groebner_error is defined.

Example:

```
  ring r=0,(a,b,c,d),lp;
option(prot);
ideal i=a+b+c+d,ab+ad+bc+cd,abc+abd+acd+bcd,abcd-1; // cyclic 4
groebner(i);
↦ std in (0),(a,b,c,d,@t),(dp,C)
↦ [63:1]1(3)s2(2)s3s4-s5ss6-s7--
↦ product criterion:8 chain criterion:5
↦ std with hilb in  (0),(a,b,c,d,@t),(lp(4),C)
↦ [63:1]1(3)s2(2)s3s4-s5ss6shhhh8shh
↦ product criterion:9 chain criterion:8
↦ hilbert series criterion:6
↦ dehomogenization
↦ imap to original ring
↦ simplification
↦ _[1]=c2d6-c2d2-d4+1
↦ _[2]=c3d2+c2d3-c-d
↦ _[3]=bd4-b+d5-d
↦ _[4]=bc-bd5+c2d4+cd-d6-d2
↦ _[5]=b2+2bd+d2
↦ _[6]=a+b+c+d
ring rp=(0,a,b),(c,d), lp;
```

```
ideal i=imap(r,i);
ideal j=groebner(i);
↦ std in 0,(c,d,a,b,@t),(dp,C)
↦ [63:1]1(3)s2(2)s3s4-s5ss6-s7--
↦ product criterion:8 chain criterion:5
↦ std with hilb in  (0),(c,d,a,b,@t),(lp(2),C, dp(3))
↦ [63:3]1(3)s2(2)s3s4-s5ss6shhhh8shh
↦ product criterion:9 chain criterion:8
↦ hilbert series criterion:6
↦ dehomogenization
↦ imap to original ring
↦ simplification
option(noprot);
j; simplify(j,1); std(i);
↦ j[1]=(a3b2+a2b3-a-b)
↦ _[1]=1
↦ _[1]=1
if (system("with","MP")) {groebner(i,0);}
↦ // ** groebner did not finish
↦ _[1]=0
defined(groebner_error);
↦ 1
```

See also: Section 5.1.120 [std], page 202; Section 5.1.121 [stdfglm], page 204; Section 5.1.122 [stdhilb], page 204.

5.1.43 help

Syntax: `help;`
`help` topic ;

Type: none

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

Note:

- `?` may be used instead of `help`.
- `topic` can be an index entry of the SINGULAR manual or the name of a (loaded) procedure which has a help section.
- `topic` may contain wildcard characters (i.e., `*` characters).
- If a (possibly "wildcarded") `topic` cannot be found (or uniquely matched) a warning is displayed and no help information is provided.
- If `topic` is the name of a (loaded) procedure whose help section has changed w.r.t. the help available in the manual then, instead of displaying the respective help section of the manual in the help browser, the "newer" help section of the procedure is simply printed to the terminal.
- The browser in which the help information is displayed can be either set with the command-line option `--browser=<browser>` (see Section 3.1.6 [Command line options], page 18), or with the command `system("--browser", "<browser>")`. Use the command `system("browsers");` for a list of all available browsers. See Section 3.1.3 [The online help system], page 14, for more details about help browsers.

Example:

```
help;      // display title page of manual
help ring; // display help for 'ring'
?ringe;    // equivalent to 'help ringe;'
↦ // ** No help for topic 'ringe' (not even for '*ringe*')
↦ // ** Try '?;'       for general help
↦ // ** or  '?Index;'  for all available help topics
?ring*;
↦ //  ** No unique help for 'ring*'
↦ //  ** Try one of
↦ ?Rings and orderings; ?Rings and standard bases; ?ring;
↦ ?ring declarations; ?ring operations; ?ring related functions;
↦ ?ring.lib; ?ring_lib; ?ringtensor; ?ringweights;
help Rings and orderings;
help standard.lib;  // displays help for library 'standard.lib'
```

See Section 3.1.6 [Command line options], page 18; Section 3.8.2 [Format of a library], page 52; Section 3.7.1 [Procedure definition], page 47; Section 3.1.3 [The online help system], page 14; Section 5.1.124 [system], page 205.

5.1.44 highcorner

Syntax: `highcorner` (ideal_expression)
`highcorner` (module_expression)

Type: poly, resp. vector

Purpose: returns the smallest monomial not contained in the ideal, resp. module, generated by the initial terms of the given generators. If the generators are a standard basis, this is also the smallest monomial not contained in the ideal, resp. module.
If the ideal, resp. module, is not zero-dimensional, 0 is returned.

Note: Let the ideal I be given by a standard basis. Then `highcorner(I)` returns 0 iff `dim(I)>0` or `dim(I)=-1`. Otherwise it returns the smallest monomial m not in I which has the following properties (with x_i the variables of the basering):

- if $x_i > 1$ then x_i does not divide m (e.g., m=1 if the ordering is global)
- given any set of generators $f_1, \ldots, f_k$ of I, let f_i' be obtained from f_i by deleting the terms divisible by $x_i \cdot m$ for all i with $x_i < 1$. Then $f_1', \ldots, f_k'$ generate I.

Example:

```
ring r=0,(x,y),ds;
ideal i=x3,x2y,y3;
highcorner(std(i));
↦ xy2
highcorner(std(ideal(1)));
↦ 0
```

See Section 5.1.19 [dim], page 134; Section 5.1.120 [std], page 202; Section 5.1.134 [vdim], page 212.

5.1.45 hilb

Syntax: `hilb` (ideal_expression)
`hilb` (module_expression)
`hilb` (ideal_expression, int_expression)

`hilb` (module_expression, int_expression)
`hilb` (ideal_expression, int_expression , intvec_expression)
`hilb` (module_expression, int_expression , intvec_expression)

Type: none (if called with one argument)
intvec (if called with two or three arguments)

Purpose: computes the (weighted) Hilbert series of the ideal, resp. module, defined by the leading terms of the generators of the given ideal, resp. module.
If `hilb` is called with one argument, then the 1st and 2nd Hilbert series together with some additional information are displayed.
If `hilb` is called with two arguments, then the n-th Hilbert series is returned as an intvec, where n=1,2 is the second argument.
If a weight vector w is a given as 3rd argument, then the Hilbert series is computed w.r.t. these weights w (by default all weights are set to 1).

Caution: The last entry of the returned intvec is not part of the actual Hilbert series, but is used in the Hilbert driven standard basis computation (see Section 5.1.122 [stdhilb], page 204).

Note: If the input is homogeneous w.r.t. the weights and a standard basis, the result is the (weighted) Hilbert series of the original ideal, resp. module.

Example:

```
  ring R=32003,(x,y,z),dp;
  ideal i=x2,y2,z2;
  ideal s=std(i);
  hilb(s);
↦ //         1 t^0
↦ //        -3 t^2
↦ //         3 t^4
↦ //        -1 t^6
↦
↦ //         1 t^0
↦ //         3 t^1
↦ //         3 t^2
↦ //         1 t^3
↦ // dimension   = 0
↦ // degree      = 0
  hilb(s,1);
↦ 1,0,-3,0,3,0,-1,0
  hilb(s,2);
↦ 1,3,3,1,0
  intvec w=2,2,2;
  hilb(s,1,w);
↦ 1,0,0,0,-3,0,0,0,3,0,0,0,-1,0
```

See Section C.2 [Hilbert function], page 281; Section 4.2 [ideal], page 68; Section 4.5 [intvec], page 81; Section 4.10 [module], page 100; Section 5.1.120 [std], page 202; Section 5.1.122 [stdhilb], page 204.

5.1.46 homog

Syntax: `homog` (ideal_expression)
`homog` (module_expression)

Type: int

Purpose: tests for homogeneity: returns 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 4.2 [ideal], page 68; Section 4.10 [module], page 100; Section 4.12 [poly], page 107; Section 4.18 [vector], page 118.

5.1.47 hres

Syntax: `hres` (ideal_expression`,` int_expression)

Type: resolution

Purpose: computes a free resolution of an ideal using the Hilbert-driven algorithm.

More precisely, let R be the basering and I be the given ideal. Then `hres` computes a minimal free resolution of R/I

$$... \longrightarrow F_2 \xrightarrow{A_2} F_1 \xrightarrow{A_1} R \longrightarrow R/I \longrightarrow 0.$$

If the int_expression k is not zero then the computation stops after k steps and returns a list of modules $M_i = \texttt{module}(A_i)$, i=1..k.

`list L=hres(I,0);` returns a list L of n modules (where n is the number of variables of the basering) such that $\texttt{L[i]} = M_i$ in the above notation.

Note: The ideal_expression has to be homogeneous.
Accessing single elements of a resolution may require that some partial computations have to be finished and may therefore take some time.

Example:

```
  ring r=0,(x,y,z),dp;
  ideal I=xz,yz,x3-y3;
  def L=hres(I,0);
  print(betti(L),"betti");
↦            0     1     2
↦ ------------------------
↦     0:     1     -     -
↦     1:     -     2     1
↦     2:     -     1     1
↦ ------------------------
↦ total:     1     3     2
  L[2];     // the first syzygy module of r/I
↦ _[1]=-x*gen(1)+y*gen(2)
↦ _[2]=-x2*gen(2)+y2*gen(1)+z*gen(3)
```

See Section 5.1.3 [betti], page 124; Section 4.2 [ideal], page 68; Section 4.3 [int], page 72; Section 5.1.69 [lres], page 165; Section 5.1.74 [minres], page 168; Section 4.10 [module], page 100; Section 5.1.78 [mres], page 171; Section 5.1.109 [res], page 193; Section 5.1.118 [sres], page 200.

5.1.48 imap

Syntax: `imap` (ring_name, name)

Type: number, poly, vector, ideal, module, matrix or list (the same type as the second argument)

Purpose: identity map on common subrings. `imap` is the map between rings and qrings with compatible ground fields which is the identity on variables and parameters of the same name and 0 otherwise. (See Section 4.8 [map], page 94 for a description of possible mapping between different ground fields). Useful for mapping from a homogenized ring to the original ring or for mappings from/to rings with/without parameters. Compared with `fetch`, `imap` uses the names of variables and parameters. Unlike `map` and `fetch` `imap` can map parameters to variables.

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 5.1.31 [fetch], page 141; Section 5.1.46 [homog], page 150; Section 4.8 [map], page 94; Section 4.14 [qring], page 111; Section 4.16 [ring], page 113.

5.1.49 impart

Syntax: `impart` (number_expression)

Type: number

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

Example:

```
  ring r=(complex,i),x,dp;
  impart(1+2*i);
↦ 2
```

See Section 5.1.108 [repart], page 193.

5.1.50 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, `x(i)`, is an independent variable. Hence, the set U consisting of all variables `x(i)` with `v[i]=1` is a maximal independent set.

Note: U is a set of independent variables for I if and only if $I \cap K[U] = (0)$, i.e., eliminating the remaining variables gives (0). U is maximal if dim(I)=#U.

Syntax: `indepSet` (ideal_expression, int_expression)

Type: list

Purpose: computes a list of all maximal independent sets of the leading ideal (if the flag is 0), resp. of all those sets of independent variables of the leading ideal which cannot be enlarged.

Example:

```
  ring r=32003,(x,y,u,v,w),dp;
  ideal I=xyw,yvw,uyw,xv;
  attrib(I,"isSB",1);
  indepSet(I);
↦ 1,1,1,0,0
  eliminate(I,vw);
↦ _[1]=0
  indepSet(I,0);
↦ [1]:
↦    1,1,1,0,0
↦ [2]:
↦    0,1,1,1,0
↦ [3]:
↦    1,0,1,0,1
↦ [4]:
↦    0,0,1,1,1
  indepSet(I,1);
↦ [1]:
↦    1,1,1,0,0
↦ [2]:
```

```
↦    0,1,1,1,0
↦ [3]:
↦    1,0,1,0,1
↦ [4]:
↦    0,0,1,1,1
↦ [5]:
↦    0,1,0,0,1
  eliminate(I,xuv);
↦ _[1]=0
```

See Section 4.2 [ideal], page 68; Section 5.1.120 [std], page 202.

5.1.51 insert

Syntax: `insert` (list_expression, expression)
`insert` (list_expression, expression, int_expression)

Type: list

Purpose: inserts a new element (expression) into a list at the beginning, or (if called with 3 arguments) after the given position (the input is not changed).

Example:

```
  list L=1,2;
  insert(L,4,2);
↦ [1]:
↦    1
↦ [2]:
↦    2
↦ [3]:
↦    4
  insert(L,4);
↦ [1]:
↦    4
↦ [2]:
↦    1
↦ [3]:
↦    2
```

See Section 5.1.16 [delete], page 132; Section 4.7 [list], page 91.

5.1.52 interred

Syntax: `interred` (ideal_expression)
`interred` (module_expression)

Type: the same as the input type

Purpose: interreduces a set of polynomials/vectors.
input: $f_1, \ldots, f_n$
output: $g_1, \ldots, g_s$ with $s \leq n$ and the properties

- $(f_1, \ldots, f_n) = (g_1, \ldots, g_s)$
- $L(g_i) \neq L(g_j)$ for all $i \neq j$
- in the case of a global ordering (polynomial ring):
  $L(g_i)$ does not divide m for all monomials m of $\{g_1, \ldots, g_{i-1}, g_{i+1}, \ldots, g_s\}$

- in the case of a local or mixed ordering (localization of polynomial ring): if $L(g_i)|L(g_j)$ for any $i \neq j$, then $ecart(g_i) > ecart(g_j)$

Here, $L(g)$ denotes the leading term of g and $ecart(g) := deg(g) - deg(L(g))$.

Example:

```
  ring r=0,(x,y,z),dp;
  ideal i=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+xy
↦ _[2]=xy-y3
↦ _[3]=x2y-y3
```

See Section 4.2 [ideal], page 68; Section 4.10 [module], page 100; Section 5.1.120 [std], page 202.

5.1.53 intersect

Syntax: `intersect` (expression_list of ideal_expression)
`intersect` (expression_list of module_expression)

Type: ideal, resp. module

Purpose: computes the intersection of ideals, resp. modules.

Note: If the option `returnSB` 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 68; Section 4.10 [module], page 100; Section 5.1.89 [option], page 177.

5.1.54 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=x2+y3+z5;
  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 5.1.18 [diff], page 133; Section 4.2 [ideal], page 68; Section 4.10 [module], page 100; Section 5.1.87 [nvars], page 176.

5.1.55 jet

Syntax: `jet` (poly_expression, int_expression)
`jet` (vector_expression, int_expression)
`jet` (ideal_expression, int_expression)
`jet` (module_expression, int_expression)
`jet` (poly_expression, int_expression, intvec_expression)
`jet` (vector_expression, int_expression, intvec_expression)
`jet` (ideal_expression, int_expression, intvec_expression)
`jet` (module_expression, int_expression, intvec_expression)
`jet` (poly_expression, int_expression, poly_expression)
`jet` (vector_expression, int_expression, poly_expression)
`jet` (ideal_expression, int_expression, matrix_expression)
`jet` (module_expression, int_expression, matrix_expression)

Type: the same as the type of the first argument

Purpose: deletes from the first argument all terms of degree bigger than the second argument.
If a third argument `w` of type intvec is given, the degree is replaced by the weighted degree defined by `w`.
If a third argument `u` of type poly or matrix is given, the first argument `p` is replaced by `p/u`.

Example:

```
  ring r=32003,(x,y,z),(c,dp);
  jet(1+x+x2+x3+x4,3);
↦ x3+x2+x+1
  poly f=1+x+x2+xz+y2+x3+y3+x2y2+z4;
  jet(f,3);
↦ x3+y3+x2+y2+xz+x+1
  intvec iv=2,1,1;
  jet(f,3,iv);
```

```
↦ y3+y2+xz+x+1
  // the part of f with (total) degree >3:
  f-jet(f,3);
↦ x2y2+z4
  // the homogeneous part of f of degree 2:
  jet(f,2)-jet(f,1);
↦ x2+y2+xz
  // the part of maximal degree:
  jet(f,deg(f))-jet(f,deg(f)-1);
↦ x2y2+z4
  // the absolute term of f:
  jet(f,0);
↦ 1
  // now for other types:
  ideal i=f,x,f*f;
  jet(i,2);
↦ _[1]=x2+y2+xz+x+1
↦ _[2]=x
↦ _[3]=3x2+2y2+2xz+2x+1
  vector v=[f,1,x];
  jet(v,1);
↦ [x+1,1,x]
  jet(v,0);
↦ [1,1]
  v=[f,1,0];
  module m=v,v,[1,x2,z3,0,1];
  jet(m,2);
↦ _[1]=[x2+y2+xz+x+1,1]
↦ _[2]=[x2+y2+xz+x+1,1]
↦ _[3]=[1,x2,0,0,1]
```

See Section 5.1.14 [deg], page 131; Section 4.2 [ideal], page 68; Section 4.3 [int], page 72; Section 4.5 [intvec], page 81; Section 4.10 [module], page 100; Section 4.12 [poly], page 107; Section 4.18 [vector], page 118.

5.1.56 kbase

Syntax: `kbase` (ideal_expression)
`kbase` (module_expression)
`kbase` (ideal_expression`,` int_expression)
`kbase` (module_expression`,` int_expression)

Type: the same as the input type of the first argument

Purpose: with one argument: computes a vector space basis (consisting of monomials) of the quotient ring by the ideal, resp. of a free module by the module, in case it is finite dimensional and if the input is a standard basis with respect to the ring ordering. If the input is not a standard basis, the leading terms of the input are used and the result may have no meaning.
With two arguments: computes the part of a vector space basis of the respective quotient with degree of the monomials equal to the second argument. Here, the quotient does not need to be finite dimensional.

Example:

```
  ring r=32003,(x,y,z),ds;
  ideal i=x2,y2,z;
  kbase(std(i));
↦ _[1]=xy
↦ _[2]=y
↦ _[3]=x
↦ _[4]=1
  i=x2,y3,xyz;  // quotient not finite dimensional
  kbase(std(i),2);
↦ _[1]=z2
↦ _[2]=yz
↦ _[3]=xz
↦ _[4]=y2
↦ _[5]=xy
```

See Section 4.2 [ideal], page 68; Section 4.10 [module], page 100; Section 5.1.134 [vdim], page 212.

5.1.57 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 line 7: `  i;'
  listvar();
↦ // LIB                  [0]  string standard.lib
```

See Section 5.1.13 [defined], page 131; Section D.2.2 [general_lib], page 294; Section 5.1.82 [names], page 173.

5.1.58 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 121; Section 5.1.89 [option], page 177.

5.1.59 koszul

Syntax: `koszul` (int_expression, int_expression)
`koszul` (int_expression, ideal_expression)
`koszul` (int_expression, int_expression, ideal_expression)

Type: matrix

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

`koszul(d,id)` computes a matrix of the Koszul relations of degree d of the generators of the ideal `id`.

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

Note: `koszul(1,id),koszul(2,id),...` form a complex, that is, the product of the matrices `koszul(i,id)` and `koszul(i+1,id)` equals zero.

Example:

```
  ring r=32003,(x,y,z),dp;
  print(koszul(2,3));
↦ -y,-z,0,
↦ x, 0, -z,
↦ 0, x, y
  ideal I=xz2+yz2+z3,xyz+y2z+yz2,xy2+y3+y2z;
  print(koszul(1,I));
↦ xz2+yz2+z3,xyz+y2z+yz2,xy2+y3+y2z
  print(koszul(2,I));
↦ -xyz-y2z-yz2,-xy2-y3-y2z,0,
↦ xz2+yz2+z3,  0,          -xy2-y3-y2z,
↦ 0,           xz2+yz2+z3, xyz+y2z+yz2
  print(koszul(2,I)*koszul(3,I));
↦ 0,
↦ 0,
↦ 0
```

See Section 4.3 [int], page 72; Section 4.9 [matrix], page 96.

5.1.60 laguerre

Syntax: `laguerre` (poly_expression, int_expression, int_expression)

Type: list

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

Note: If the ground field is the field of complex numbers, the elements of the list are of type number, otherwise of type string.

Example:

```
ring rs1=0,(x,y),lp;
poly f=15x5+x3+x2-10;
laguerre(f,10,2);
↦ [1]:
↦    (0.2930464644-I*0.9003002396)
↦ [2]:
↦    (0.2930464644+I*0.9003002396)
↦ [3]:
↦    (-0.7392783383-I*0.5355190078)
↦ [4]:
↦    (-0.7392783383+I*0.5355190078)
↦ [5]:
↦    0.8924637479
```

5.1.61 lead

Syntax: `lead` (poly_expression)
`lead` (vector_expression)
`lead` (ideal_expression)
`lead` (module_expression)

Type: the same as the input type

Purpose: returns the leading (or initial) term(s) of a polynomial, a vector, resp. of the generators of an ideal or module with respect to the monomial ordering.

Note: `IN` may be used instead of `lead`.

Example:

```
  ring r=32003,(x,y,z),(c,ds);
  poly f=2x2+3y+4z3;
  vector v=[2x10,f];
  ideal i=f,z;
  module m=v,[0,0,2+x];
  lead(f);
↦ 3y
  lead(v);
↦ [2x10]
  lead(i);
↦ _[1]=3y
↦ _[2]=z
  lead(m);
↦ _[1]=[2x10]
↦ _[2]=[0,0,2]
  lead(0);
↦ 0
```

See Section 4.2 [ideal], page 68; Section 5.1.62 [leadcoef], page 161; Section 5.1.63 [leadexp], page 161; Section 5.1.64 [leadmonom], page 162; Section 4.10 [module], page 100; Section 4.12 [poly], page 107; Section 4.18 [vector], page 118.

5.1.62 leadcoef

Syntax: `leadcoef` (poly_expression)
`leadcoef` (vector_expression)

Type: number

Purpose: returns the leading (or initial) coefficient of a polynomial or a vector with respect to the monomial ordering.

Example:

```
  ring r=32003,(x,y,z),(c,ds);
  poly f=x2+y+z3;
  vector v=[2*x^10,f];
  leadcoef(f);
↦ 1
  leadcoef(v);
↦ 2
  leadcoef(0);
↦ 0
```

See Section 5.1.61 [lead], page 160; Section 5.1.63 [leadexp], page 161; Section 5.1.64 [leadmonom], page 162; Section 4.11 [number], page 103; Section 4.12 [poly], page 107; Section 4.18 [vector], page 118.

5.1.63 leadexp

Syntax: `leadexp` (poly_expression)
`leadexp` (vector_expression)

Type: intvec

Purpose: returns the exponent vector of the leading monomial of a polynomial or a vector. In the case of a vector the last component is the index in the vector.

Example:

```
  ring r=32003,(x,y,z),(c,ds);
  poly f=x2+y+z3;
  vector v=[2*x^10,f];
  leadexp(f);
↦ 0,1,0
  leadexp(v);
↦ 10,0,0,1
  leadexp(0);
↦ 0,0,0
```

See Section 4.5 [intvec], page 81; Section 5.1.61 [lead], page 160; Section 5.1.62 [leadcoef], page 161; Section 5.1.64 [leadmonom], page 162; Section 4.12 [poly], page 107; Section 4.18 [vector], page 118.

5.1.64 leadmonom

Syntax: `leadmonom` (poly_expression)
`leadmonom` (vector_expression)

Type: the same as the input type

Purpose: returns the leading monomial of a polynomial or a vector as a polynomial or vector whose coefficient is one.

Example:

```
  ring r=32003,(x,y,z),(c,ds);
  poly f=2x2+3y+4z3;
  vector v=[2x10,f];
  leadmonom(f);
↦ y
  leadmonom(v);
↦ [x10]
  leadmonom(0);
↦ 0
```

See Section 4.5 [intvec], page 81; Section 5.1.61 [lead], page 160; Section 5.1.62 [leadcoef], page 161; Section 5.1.63 [leadexp], page 161; Section 4.12 [poly], page 107; Section 4.18 [vector], page 118.

5.1.65 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 `/` and cannot be located in the current directory, each directory contained in the library `SearchPath` is searched for file of this name. See Section 3.8.1 [Loading of a library], page 51, for more info on `SearchPath`.

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

Syntax: `LIB`;

Type: string

Purpose: shows all loaded libraries written in Singular.

Example:

```
  option(loadLib); // show loading of libraries
  LIB;             // standard.lib is loaded
↦ standard.lib

                   // the names of the procedures of inout.lib
  LIB "inout.lib"; // are now known to Singular
↦ // ** loaded inout.lib (1.21.2.3,2002/02/20)
  LIB;
↦ standard.lib,inout.lib
```

See Section 3.1.6 [Command line options], page 18; Section 3.8.1 [Loading of a library], page 51; Section 2.3.3 [Procedures and libraries], page 9; Appendix D [SINGULAR libraries], page 293; Section 4.13 [proc], page 110; Section D.1 [standard_lib], page 293; Section 4.17 [string], page 114; Section 5.1.124 [system], page 205.

5.1.66 lift

Syntax: `lift` (ideal_expression`,` subideal_expression)
`lift` (module_expression`,` submodule_expression)
`lift` (ideal_expression`,` subideal_expression`,` matrix_name)
`lift` (module_expression`,` submodule_expression`,` matrix_name)

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

Note: Gives a warning if `sm` is not a submodule.

Example:

```
  ring r=32003,(x,y,z),(dp,C);
  ideal m=3x2+yz,7y6+2x2y+5xz;
  poly f=y7+x3+xyz+z2;
  ideal i=jacob(f);
  matrix T=lift(i,m);
  matrix(m)-matrix(i)*T;
↦ _[1,1]=0
↦ _[1,2]=0
```

See Section 5.1.20 [division], page 134; Section 4.2 [ideal], page 68; Section 4.10 [module], page 100.

5.1.67 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.
That is, if `m` is the ideal or module, `sm` the standard basis returned by `liftstd`, and `T` the transformation matrix then `matrix(sm)=matrix(m)*T` and `sm=ideal(matrix(m)*T)`, resp. `sm=module(matrix(m)*T)`.

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 68; Section 4.9 [matrix], page 96; Section 5.1.89 [option], page 177; Section 4.16 [ring], page 113; Section 5.1.120 [std], page 202.

5.1.68 listvar

Syntax: `listvar` ([package])
`listvar` ([package,] type)
`listvar` ([package,] ring_name)
`listvar` ([package,] name)
`listvar` ([package,] `all`)

Type: none

Purpose: lists all (user-)defined names in the current namespace:

- `listvar()`: all currently visible names except procedures,
- `listvar`(type): all currently visible names of the given type,
- `listvar`(ring_name): all names which belong to the given ring,
- `listvar`(name): the object with the given name,
- `listvar(all)`: all names except 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
↦ // fprintf              [0]  proc from standard.lib
↦ // printf               [0]  proc from standard.lib
↦ // sprintf              [0]  proc from standard.lib
↦ // 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 3.5.3 [Names], page 41; Section 3.7.2 [Names in procedures], page 49; Section 5.1.13 [defined], page 131; Section 5.1.82 [names], page 173; Section 5.1.128 [type], page 209.

5.1.69 lres

Syntax: `lres` (ideal_expression, int_expression)

Type: resolution

Purpose: computes a free resolution of an ideal using La Scala's algorithm.

More precisely, let R be the basering and I be the given ideal. Then `lres` computes a minimal free resolution of R/I

$$... \longrightarrow F_2 \xrightarrow{A_2} F_1 \xrightarrow{A_1} R \longrightarrow R/I \longrightarrow 0.$$

If the int_expression k is not zero then the computation stops after k steps and returns a list of modules $M_i = \texttt{module}(A_i)$, i=1..k.

list L=lres(I,0); returns a list L of n modules (where n is the number of variables of the basering) such that L[i] $= M_i$ in the above notation.

Note: The ideal_expression has to be homogeneous.
Accessing single elements of a resolution may require that some partial computations have to be finished and may therefore take some time.

Example:

```
  ring r=0,(x,y,z),dp;
  ideal I=xz,yz,x3-y3;
  def L=lres(I,0);
  print(betti(L),"betti");
↦            0     1     2
↦ ------------------------
↦     0:     1     -     -
↦     1:     -     2     1
↦     2:     -     1     1
↦ ------------------------
↦ total:     1     3     2
  L[2];     // the first syzygy module of r/I
↦ _[1]=-x*gen(1)+y*gen(2)
↦ _[2]=-x2*gen(2)+y2*gen(1)+z*gen(3)
```

See Section 5.1.3 [betti], page 124; Section 5.1.47 [hres], page 151; Section 4.2 [ideal], page 68; Section 4.3 [int], page 72; Section 5.1.74 [minres], page 168; Section 4.10 [module], page 100; Section 5.1.78 [mres], page 171; Section 5.1.109 [res], page 193; Section 5.1.118 [sres], page 200.

5.1.70 maxideal

Syntax: maxideal (int_expression)

Type: ideal

Purpose: returns the power given by int_expression of the maximal ideal generated by all ring variables (maxideal(i)=1 for i<=0).

Example:

```
  ring r=32003,(x,y,z),dp;
  maxideal(2);
↦ _[1]=z2
↦ _[2]=yz
↦ _[3]=y2
↦ _[4]=xz
↦ _[5]=xy
↦ _[6]=x2
```

See Section 4.2 [ideal], page 68; Section 4.16 [ring], page 113.

5.1.71 memory

Syntax: memory (int_expression)

Type: int

Purpose: returns statistics concerning the memory management:

- memory(0) is the number of active (used) bytes,

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

Note: To monitor the memory usage during ongoing computations the option `mem` should be set (using the command `option(mem);`, see also Section 5.1.89 [option], page 177).

Example:

```
  ring r=0,(x(1..500)),dp;
  poly p=(x(1)+x(500))^50;
  proc ReportMemoryUsage()
  {  "Memory currently used by SINGULAR     :",memory(0),"Byte (",
     memory(0)/1023, "KByte)" +newline+
     "Memory currently allocated from system:",memory(1), "Byte (",
     memory(1)/1023, "KByte)";
     "Maximal memory allocated from system  :",memory(2), "Byte (",
     memory(2)/1023, "KByte)";
  }
  ReportMemoryUsage();
↦ Memory currently used by SINGULAR     : 205408 Byte ( 200 KByte)
↦ Memory currently allocated from system: 671132 Byte ( 656 KByte)
↦ Maximal memory allocated from system  : 671132 Byte ( 656 KByte)
  kill p;
  ReportMemoryUsage(); // less memory used: p killed
↦ Memory currently used by SINGULAR     : 152172 Byte ( 148 KByte)
↦ Memory currently allocated from system: 671132 Byte ( 656 KByte)
↦ Maximal memory allocated from system  : 671132 Byte ( 656 KByte)
  kill r;
  ReportMemoryUsage(); // even less memory: r killed
↦ Memory currently used by SINGULAR     : 141592 Byte ( 138 KByte)
↦ Memory currently allocated from system: 671132 Byte ( 656 KByte)
↦ Maximal memory allocated from system  : 671132 Byte ( 656 KByte)
```

See Section 5.1.89 [option], page 177; Section 5.1.124 [system], page 205.

5.1.72 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 5.1.79 [mstd], page 171.

5.1.73 minor

Syntax: `minor` (matrix_expression, int_expression)
`minor` (matrix_expression, int_expression, ideal_expression)

Type: ideal

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

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)
  minor(m,2,std(ideal(x(1))));
↦ _[1]=-x(4)^2+x(3)*x(5)
↦ _[2]=-x(3)*x(4)+x(2)*x(5)
↦ _[3]=-x(2)*x(4)
↦ _[4]=x(3)^2-x(2)*x(4)
↦ _[5]=x(2)*x(3)
↦ _[6]=-x(2)^2
```

See Section 5.1.17 [det], page 133.

5.1.74 minres

Syntax: `minres` (list_expression)

Type: list

Syntax: `minres` (resolution_expression)

Type: resolution

Purpose: minimizes a free resolution of an ideal or module given by the list_expression, resp. resolution_expression.

Example:

```
  ring r1=32003,(x,y),dp;
  ideal i=x5+xy4,x3+x2y+xy2+y3;
  resolution rs=lres(i,0);
  rs;
↦   1       2       1
↦ r1 <--  r1 <--  r1
↦
↦ 0       1       2
↦ 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
↦ r1 <--  r1 <--  r1
↦
↦ 0       1       2
↦
  list(rs);
↦ [1]:
↦    _[1]=x3+x2y+xy2+y3
↦    _[2]=xy4
↦ [2]:
↦    _[1]=xy4*gen(1)-x3*gen(2)-x2y*gen(2)-xy2*gen(2)-y3*gen(2)
```

See Section 5.1.78 [mres], page 171; Section 5.1.109 [res], page 193; Section 5.1.118 [sres], page 200.

5.1.75 modulo

Syntax: `modulo` (ideal_expression, ideal_expression)
`modulo` (module_expression, module_expression)

Type: module

Purpose: `modulo(h1,h2)` represents $h_1/(h_1 \cap h_2) \cong (h_1+h_2)/h_2$ where h_1 and h_2 are considered as submodules of the same free module R^l (l=1 for ideals). Let H_1, resp. H_2, be the matrices of size $l \times k$, resp. $l \times m$, having the generators of h_1, resp. h_2, as columns. Then $h_1/(h_1 \cap h_2) \cong R^k/ker(\overline{H_1})$ where $\overline{H_1} : R^k \rightarrow R^l/Im(H_2) = R^l/h_2$ 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 5.1.125 [syz], page 207.

5.1.76 monitor

Syntax: `monitor` (string_expression)
`monitor` (string_expression, string_expression)

Type: none

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

Example:

```
monitor("doe.tmp","io"); // log input and output to doe.tmp
ring r;
poly f=x+y+z;
int i=7;
ideal I=f,x,y;
monitor("");             // stop logging
```

5.1.77 mpresmat

Syntax: `mpresmat` (ideal_expression, int_expression)

Type: module

Purpose: computes the multipolynomial resultant matrix of the input system. Uses the sparse resultant matrix method of Gelfand, Kapranov and Zelevinsky (second parameter = 0) or the resultant matrix method of Macaulay (second parameter = 1).

Note: When using the resultant matrix method of Macaulay the input system must be homogeneous. The number of elements in the input system must be the number of variables in the basering plus one.

Example:

```
  ring rsq=(0,s,t,u),(x,y),lp;
  ideal i=s+tx+uy,x2+y2-10,x2+xy+2y2-16;
  module m=mpresmat(i,0);
  print(m);
↦ -16,0,  -10,0,  (s),0,  0,  0,  0,  0,
↦ 0,  -16,0,  -10,(u),(s),0,  0,  0,  0,
↦ 2,  0,  1,  0,  0,  (u),0,  0,  0,  0,
↦ 0,  2,  0,  1,  0,  0,  0,  0,  0,  0,
↦ 0,  0,  0,  0,  (t),0,  -10,(s),0,  -16,
↦ 1,  0,  0,  0,  0,  (t),0,  (u),(s),0,
↦ 0,  1,  0,  0,  0,  0,  1,  0,  (u),2,
↦ 1,  0,  1,  0,  0,  0,  0,  (t),0,  0,
↦ 0,  1,  0,  1,  0,  0,  0,  0,  (t),1,
↦ 0,  0,  0,  0,  0,  0,  1,  0,  0,  1
```

See Section 5.1.130 [uressolve], page 210.

5.1.78 mres

Syntax: `mres` (ideal_expression`,` int_expression)
`mres` (module_expression`,` int_expression)

Type: resolution

Purpose: computes a minimal free resolution of an ideal or module M with the standard basis method. More precisely, let A=`matrix`(M), then `mres` computes a free resolution of $coker(A) = F_0/M$

$$... \longrightarrow F_2 \stackrel{A_2}{\longrightarrow} F_1 \stackrel{A_1}{\longrightarrow} F_0 \longrightarrow F_0/M \longrightarrow 0,$$

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

Note: Accessing single elements of a resolution may require that some partial computations have to be finished and may therefore take some time.

Example:

```
  ring r=31991,(t,x,y,z,w),ls;
  ideal M=t2x2+tx2y+x2yz,t2y2+ty2z+y2zw,
          t2z2+tz2w+xz2w,t2w2+txw2+xyw2;
  resolution L=mres(M,0);
  L;
↦  1      4      15      18      7      1
↦ r <--  r <--  r <--   r <--   r <--  r
↦
↦ 0      1      2       3       4      5
↦
  // projective dimension of M is 5
```

See Section 5.1.47 [hres], page 151; Section 4.2 [ideal], page 68; Section 5.1.69 [lres], page 165; Section 4.10 [module], page 100; Section 5.1.109 [res], page 193; Section 5.1.118 [sres], page 200.

5.1.79 mstd

Syntax: `mstd` (ideal_expression)
`mstd` (module_expression)

Type: list

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

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 68; Section 5.1.72 [minbase], page 167; Section 4.10 [module], page 100; Section 5.1.120 [std], page 202.

5.1.80 mult

Syntax: `mult` (ideal_expression)
`mult` (module_expression)

Type: int

Purpose: computes the degree of the monomial ideal, resp. module, generated by the leading monomials of the input.
If the input is a standard basis of a homogeneous ideal then it returns the degree of this ideal.
If the input is a standard basis of an ideal in a (local) ring with respect to a local degree ordering then it returns the multiplicity of the ideal (in the sense of Samuel, with respect to the maximal ideal).

Example:

```
ring r=32003,(x,y),ds;
poly f=(x3+y5)^2+x2y7;
ideal i=std(jacob(f));
mult(i);
↦ 46
mult(std(f));
↦ 6
```

See Section 5.1.15 [degree], page 132; Section 5.1.19 [dim], page 134; Section 4.2 [ideal], page 68; Section 5.1.120 [std], page 202; Section 5.1.134 [vdim], page 212.

5.1.81 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 line 7: `  basering;`
  ring r;
  nameof(basering);
↦ r
```

See Section 5.1.82 [names], page 173; Section 5.1.110 [reservedName], page 194; Section 5.1.129 [typeof], page 209.

5.1.82 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, which belong to the given ring.

Example:

```
  int i=9;
  ring r;
  poly f;
  poly g;
  setring r;
  names();
↦ [1]:
↦    r
↦ [2]:
↦    i
↦ [3]:
↦    fprintf
```

```
↦ [4]:
↦    printf
↦ [5]:
↦    sprintf
↦ [6]:
↦    intersect1
↦ [7]:
↦    quotient4
↦ [8]:
↦    quotient5
↦ [9]:
↦    quotient3
↦ [10]:
↦    quotient2
↦ [11]:
↦    quotient1
↦ [12]:
↦    quotient0
↦ [13]:
↦    quot1
↦ [14]:
↦    quot
↦ [15]:
↦    res
↦ [16]:
↦    groebner
↦ [17]:
↦    stdhilb
↦ [18]:
↦    stdfglm
↦ [19]:
↦    LIB
  names(r);
↦ [1]:
↦    g
↦ [2]:
↦    f
```

See Section 5.1.81 [nameof], page 173; Section 5.1.110 [reservedName], page 194.

5.1.83 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: `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 96; Section 5.1.86 [nrows], page 176; Section 5.1.115 [size], page 197.

5.1.84 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 5.1.92 [par], page 181; Section 5.1.94 [parstr], page 182; Section 4.16 [ring], page 113.

5.1.85 nres

Syntax: `nres` (ideal_expression, int_expression)
`nres` (module_expression, int_expression)

Type: resolution

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

More precisely, let A_1=matrix(M), then `nres` computes a free resolution of $coker(A_1) = F_0/M$

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

where the columns of the matrix A_1 are the given set of generators of M. If the int expression k is not zero then the computation stops after k steps and returns a list of modules M_i = `module`(A_i), i=1..k.

`nres(M,0)` returns a list of n modules where n is the number of variables of the basering. Let `list L=nres(M,0);` then `L[1]=M` is identical to the input, `L[2]` is a minimal set of generators for the first syzygy module of `L[1]`, etc. (L[i] $= M_i$ in the notations from above).

Example:

```
  ring r=31991,(t,x,y,z,w),ls;
  ideal M=t2x2+tx2y+x2yz,t2y2+ty2z+y2zw,
          t2z2+tz2w+xz2w,t2w2+txw2+xyw2;
  resolution L=nres(M,0);
```

```
  L;
↦  1      4      15      18      7      1
↦ r <--  r <--  r <--   r <--   r <--  r
↦
↦ 0      1      2       3       4      5
↦ resolution not minimized yet
↦
```

See Section 5.1.47 [hres], page 151; Section 4.2 [ideal], page 68; Section 5.1.69 [lres], page 165; Section 4.10 [module], page 100; Section 5.1.78 [mres], page 171; Section 5.1.109 [res], page 193; Section 4.15 [resolution], page 112; Section 5.1.118 [sres], page 200.

5.1.86 nrows

Syntax: `nrows` (matrix_expression)
`nrows` (intmat_expression)
`nrows` (intvec_expression)
`nrows` (module_expression)
`nrows` (vector_expression)

Type: int

Purpose: returns the number of rows of a matrix, an intmat or an intvec, resp. the minimal rank of a free module in which the given module or vector lives (the index of the last non-zero component).

Note: Use `ncols` to get the number of columns of a given matrix or intmat.

Example:

```
  ring R;
  matrix M[2][3];
  nrows(M);
↦ 2
  nrows(freemodule(4));
↦ 4
  module m=[0,0,1];
  nrows(m);
↦ 3
  nrows([0,x,0]);
↦ 2
```

See Section 5.1.40 [gen], page 146; Section 4.9 [matrix], page 96; Section 4.10 [module], page 100; Section 5.1.83 [ncols], page 174; Section 4.18 [vector], page 118.

5.1.87 nvars

Syntax: `nvars` (ring_name)

Type: int

Purpose: returns the number of variables of a ring.

Example:

```
  ring r=(23,t,v),(x,a(1..7)),ls;
  // the variables are x,a(1),...,a(7)
  nvars(r);
↦ 8
```

See Section 5.1.84 [npars], page 175; Section 4.16 [ring], page 113; Section 5.1.132 [var], page 211; Section 5.1.133 [varstr], page 211.

5.1.88 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 5.1.8 [close], page 127; Section 4.6 [link], page 83.

5.1.89 option

Syntax: `option ()`

Type: string

Purpose: lists all defined options.

Syntax: `option` (option_name)

Type: none

Purpose: sets an option.

Note: To disable an option, use the prefix `no`.

Syntax: `option ( get )`

Type: intvec

Purpose: dumps the state of all options to an intvec.

Syntax: `option ( set,` intvec_expression)

Type: none

Purpose: restores the state of all options from an intvec (produced by `option(get)`).

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

`none` turns off all options (including the `prompt` option).

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

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

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

`oldStd` uses a more lazy approach in std computations, which was used in Singular version before 2-0 (and which may lead to faster or slower computations, depending on the example)

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

`notRegularity`
: disables the regularity bound for `res` and `mres` (see Section 5.1.107 [regularity], page 192).

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

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

`prot` shows protocol information indicating the progress during the following computations: `facstd`, `fglm`, `groebner`, `lres`, `mres`, `minres`, `mstd`, `res`, `sres`, `std`, `stdfglm`, `stdhilb`, `syz`. 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. This option is ring dependent. By default, it is set for rings with global degree orderings and not set for all other rings.

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

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

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

The following options, which also control computations, are special, since they are not manipulated by the `option` command but by a direct assignment of a value. Reset the option by assigning the value 0; the command `option(none)` will not reset them! If there is a non-zero value assigned, the command `option()` prints the option.

`multBound`
: a multiplicity bound is set (see Section 5.3.4 [multBound], page 221).

`degBound` a degree bound is set (see Section 5.3.1 [degBound], page 220).

The last set of options controls the output of SINGULAR:

`Imap` shows the mapping of variables with the fetch and imap commands.

`loadLib` shows loading of libraries (set by 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 5.1.71 [memory], page 166).

`prompt` shows prompt (`>`, resp. `.`) if ready for input (default).

`reading` shows the number of characters read from a file.

`redefine` warns about variable redefinitions (set by default).

`usage` shows correct usage in error messages (set by default).

Example:

```
  option(prot);
  option();
↦ //options: prot redefine usage prompt
  option(notSugar);
  option();
↦ //options: prot notSugar redefine usage prompt
  option(noprot);
  option();
↦ //options: notSugar redefine usage prompt
  option(none);
  option();
↦ //options: none
  ring r=0,x,dp;
  degBound=22;
  option();
↦ //options: degBound redTail redThrough intStrategy
  intvec i=option(get);
  option(none);
  option(set,i);
  option();
↦ //options: degBound redTail redThrough intStrategy
```

The output reported on `option(prot)` has the following meaning:

`facstd`	`F`	found a new factor
		all other characters: like the output of `std` and `reduce`
`fglm`	`.`	basis monomial found
	`+`	edge monomial found
	`-`	border monomial found
`groebner`		all characters: like the output of `std`
`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`
`reduce`	`r`	reduced a leading term
	`t`	reduced a non-leading term
`res`	`[` d `]`	computations of the d -th syzygy module all other characters: like the output of `std`
`sres`	`.`	syzygy found
	`(` n `)`	n elements remaining
	`[` n `]`	finished module n
`std`	`[` m `:` n `]`	internal ring change to poly representation with exponent bound m and n words in exponent vector
	`s`	found a new element of the standard basis
	`-`	reduced a pair/S-polynomial to 0
	`.`	postponed a reduction of a pair/S-polynomial
	`h`	used Hilbert series criterion
	`H(` d `)`	found a 'highest corner' of degree d , no need to consider higher degrees
	`(` n `)`	n critical pairs are still to be reduced
	`(S:` n `)`	doing complete reduction of n elements
	d	the degree of the leading terms is currently d
`stdfglm`		all characters in first part: like the output of `std` all characters in second part: like the output of `fglm`
`stdhilb`		all characters: like the output of `std`
`syz`		all characters: like the output of `std`

See Section 5.3.1 [degBound], page 220; Section 5.3.4 [multBound], page 221; Section 5.1.120 [std], page 202.

5.1.90 ord

Syntax: `ord` (poly_expression)
`ord` (vector_expression)

Type: int

Purpose: returns the (weighted) degree of the initial term of a polynomial or a vector; the weights are the weights used for the first block of the ring ordering.

Note: `ord(0)` is `-1`.

Example:

```
  ring r=7,(x,y),wp(2,3);
  ord(0);
↦ -1
  poly f=x2+y3;  // weight on y is 3
  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 131; Section 4.12 [poly], page 107; Section 4.18 [vector], page 118.

5.1.91 ordstr

Syntax: `ordstr` (ring_name)

Type: string

Purpose: returns the description of the monomial ordering of the ring.

Example:

```
  ring r=7,(x,y),wp(2,3);
  ordstr(r);
↦ wp(2,3),C
```

See Section 5.1.6 [charstr], page 126; Section 5.1.94 [parstr], page 182; Section 4.16 [ring], page 113; Section 5.1.133 [varstr], page 211.

5.1.92 par

Syntax: `par` (int_expression)

Type: number

Purpose: `par(n);` returns the n-th parameter of the basering.

Note: 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;
  char(r);  // char to get the characteristic
↦ 0
  par(2);   // par to get the n-th parameter
↦ (b)
```

See Section 5.1.4 [char], page 125; Section 5.1.84 [npars], page 175; Section 5.1.94 [parstr], page 182; Section 4.16 [ring], page 113; Section 5.1.132 [var], page 211.

5.1.93 pardeg

Syntax: `pardeg` (number_expression)

Type: int

Purpose: returns the degree of a number considered as a polynomial in the ring parameters.

Example:

```
  ring r=(0,a,b,c),(x,y,z),dp;
  pardeg(a^2*b);
↦ 3
```

See Section 5.1.14 [deg], page 131; Section 4.11 [number], page 103; Section 4.16 [ring], page 113; Section 5.1.132 [var], page 211.

5.1.94 parstr

Syntax: `parstr` (ring_name)
`parstr` (int_expression)
`parstr` (ring_name, int_expression)

Type: string

Purpose: returns the list of parameters of the ring as a string or the name of the n-th parameter where n is given by the int_expression.
If the ring_name is omitted, the basering is used, thus `parstr(n)` is equivalent to `parstr(basering,n)`.

Example:

```
  ring r=(7,a,b,c),(x,y),wp(2,3);
  parstr(r);
↦ a,b,c
  parstr(2);
↦ b
  parstr(r,3);
↦ c
```

See Section 5.1.6 [charstr], page 126; Section 5.1.84 [npars], page 175; Section 5.1.91 [ordstr], page 181; Section 5.1.92 [par], page 181; Section 4.16 [ring], page 113; Section 5.1.133 [varstr], page 211.

5.1.95 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 4.2 [ideal], page 68; Section 4.8 [map], page 94; Section 4.16 [ring], page 113.

5.1.96 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 D.2.2 [general_lib], page 294; Section 4.3 [int], page 72.

5.1.97 print

Syntax: `print` (expression)
`print` (expression, `"betti"`)
`print` (expression, format_string)

Type: none (for the first two calling sequences), resp.
string (for the last calling sequence)

Purpose: The first form prints the expression to the terminal and has no return value. Use the format string `%p` to print into a string (see below).
The second form prints the graded Betti numbers from a matrix. See the description of the format string `"betti"` below for more details.
The last form returns the printed output as a string. The format string determines which format to use to generate the string.

The following format strings are supported:

`"betti"` The Betti numbers are printed in a matrix-like format where the entry d in row i and column j is the minimal number of generators in degree $i + j$ of the j-th syzygy module of R^n/M (the 0th and 1st syzygy module of R^n/M is R^n and M, resp.).

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

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

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

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

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

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

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

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

Example:

```
  ring r=0,(x,y,z),dp;
  module m=[1,y],[0,x+z];
  m;
↦ m[1]=y*gen(2)+gen(1)
↦ m[2]=x*gen(2)+z*gen(2)
  print(m);  // the columns generate m
↦ 1,0,
↦ y,x+z
  string s=print(m,"%s"); s;
↦ y*gen(2)+gen(1),x*gen(2)+z*gen(2)
  s=print(m,"%2s"); s;
↦ y*gen(2)+gen(1),
↦ x*gen(2)+z*gen(2)
↦
  s=print(m,"%l"); s;
↦ module(y*gen(2)+gen(1),x*gen(2)+z*gen(2))
  s=print(m,"%;"); s;
↦ m[1]=y*gen(2)+gen(1)
↦ m[2]=x*gen(2)+z*gen(2)
↦
  s=print(m,"%t"); s;
↦ // m                    [0]  module, rk 2, 2 generator(s)
↦ m[1]=y*gen(2)+gen(1)
↦ m[2]=x*gen(2)+z*gen(2)
  s=print(m,"%p"); s;
↦ 1,0,
↦ y,x+z
↦
  intmat M=betti(mres(m,0));
  print(M,"betti");
↦            0     1
↦ ------------------
↦     0:     1     1
↦ ------------------
```

```
↦ total:     1     1
  list l=r,M;
  s=print(l,"%s"); s;
↦ (0),(x,y,z),(dp(3),C),1,1
  s=print(l,"%2s"); s;
↦ (0),(x,y,z),(dp(3),C),
↦ 1,1
↦
  s=print(l,"%l"); s;
↦ list("(0),(x,y,z),(dp(3),C)",intmat(intvec(1,1 ),1,2))
```

See Section 3.5.5 [Type conversion and casting], page 42; Section 5.1.3 [betti], page 124; Section 5.1.12 [dbprint], page 130; Section 5.1.37 [fprintf], page 144; Section 5.1.98 [printf], page 185; Section 5.3.7 [short], page 222; Section 5.1.117 [sprintf], page 199; Section 4.17.3 [string type cast], page 116; Section 5.1.128 [type], page 209.

5.1.98 printf

Procedure from library `standard.lib` (see Section D.1 [standard_lib], page 293).

Syntax: `printf (` string_expression `[,` any_expressions`] )`

Return: none

Purpose: `printf(fmt,...);` performs output formatting. The first argument is a format control string. Additional arguments may be required, depending on the content of the control string. A series of output characters is generated as directed by the control string; these characters are displayed (i.e., printed to standard out).
The control string `fmt` is simply text to be copied, except that the string may contain conversion specifications.
Do `help print;` for a listing of valid conversion specifications. As an addition to the conversions of `print`, the `%n` and `%2` conversion specification does not consume an additional argument, but simply generates a newline character.

Note: If one of the additional arguments is a list, then it should be enclosed once more into a `list()` command, since passing a list as an argument flattens the list by one level.

Example:

```
  ring r=0,(x,y,z),dp;
module m=[1,y],[0,x+z];
intmat M=betti(mres(m,0));
list l=r,m,M;
printf("s:%s,l:%l",1,2);
↦ s:1,l:int(2)
printf("s:%s",l);
↦ s:(0),(x,y,z),(dp(3),C)
printf("s:%s",list(l));
↦ s:(0),(x,y,z),(dp(3),C),y*gen(2)+gen(1),x*gen(2)+z*gen(2),1,1
printf("2l:%2l",list(l));
↦ 2l:list("(0),(x,y,z),(dp(3),C)",
↦ module(y*gen(2)+gen(1),
↦ x*gen(2)+z*gen(2)),
↦ intmat(intvec(1,1 ),1,2))
↦
printf("%p",list(l));
```

```
↦ [1]:
↦    //   characteristic : 0
↦ //   number of vars : 3
↦ //        block   1 : ordering dp
↦ //                  : names    x y z
↦ //        block   2 : ordering C
↦ [2]:
↦    _[1]=y*gen(2)+gen(1)
↦    _[2]=x*gen(2)+z*gen(2)
↦ [3]:
↦    1,1
↦ 
printf("%;",list(l));
↦ [1]:
↦    //   characteristic : 0
↦ //   number of vars : 3
↦ //        block   1 : ordering dp
↦ //                  : names    x y z
↦ //        block   2 : ordering C
↦ [2]:
↦    _[1]=y*gen(2)+gen(1)
↦    _[2]=x*gen(2)+z*gen(2)
↦ [3]:
↦    1,1
↦ 
printf("%b",M);
↦            0     1
↦ ------------------
↦     0:     1     1
↦ ------------------
↦ total:     1     1
↦ 
```

See also: Section 5.1.37 [fprintf], page 144; Section 5.1.97 [print], page 183; Section 5.1.117 [sprintf], page 199; Section 4.17 [string], page 114.

5.1.99 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 100.

5.1.100 qhweight

Syntax: `qhweight` (ideal_expression)

Type: intvec

Purpose: computes the weight vector of the variables for a quasihomogeneous ideal. If the input is not weighted homogeneous, an intvec of zeros is returned.

Example:

```
  ring h1=32003,(t,x,y,z),dp;
  ideal i=x4+y3+z2;
  qhweight(i);
↦ 0,3,4,6
```

See Section 4.2 [ideal], page 68; Section 4.5 [intvec], page 81; Section 5.1.136 [weight], page 212.

5.1.101 quot

Procedure from library `standard.lib` (see Section D.1 [standard_lib], page 293).

Syntax: `quot` (module_expression, module_expression)
`quot` (module_expression, module_expression, int_expression)
`quot` (ideal_expression, ideal_expression)
`quot` (ideal_expression, ideal_expression, int_expression)

Type: ideal

Syntax: `quot` (module_expression, ideal_expression)

Type: module

Purpose: computes the quotient of the 1st and the 2nd argument. If a 3rd argument 'n' is given the n-th method is used (n=1...5).

Example:

```
  ring r=181,(x,y,z),(c,ls);
ideal id1=maxideal(4);
ideal id2=x2+xyz,y2-z3y,z3+y5xz;
option(prot);
ideal id3=quotient(id1,id2);
↦ [1023:1]4(14)s(13)s(12)s(11)s(10)s(9)s(8)s(7)s(6)s(5)s(4)s(3)s(2)sss
↦ product criterion:12 chain criterion:0
↦ {3}[1023:3]8(2)s(4)s(5)s9s.s.s.s.s.s
↦ product criterion:0 chain criterion:39
id3;
↦ id3[1]=z2
↦ id3[2]=yz
↦ id3[3]=y2
↦ id3[4]=xz
↦ id3[5]=xy
↦ id3[6]=x2
```

```
ideal id4=quot(id1,id2,1);
↦ [1023:2]4(14)s(13)s(12)s(11)s(10)s(9)s(8)s(7)s(6)s(5)s(4)s(3)s(2)sss
↦ product criterion:0 chain criterion:0
↦ (S:3)rtrtrt
↦ {1}[1023:3]4(2)s(4)s(5)s5s.s(3)s(2)sss
↦ product criterion:0 chain criterion:39
↦ {1}[1023:3]4(5)s(4)s(3)s(2)sss
↦ product criterion:0 chain criterion:9
↦ [255:3]3(11)s(10)sss(9)s(10)s(8)ss(7)s(8)s(6)s(7)s4(5)-----
↦ product criterion:12 chain criterion:9
↦ {1}[1023:3]4(2)sss
↦ product criterion:0 chain criterion:12
↦ [255:3]2(8)s(7)s(6)s3(5)ssssss4-----
↦ product criterion:15 chain criterion:3
id4;
↦ id4[1]=z2
↦ id4[2]=yz
↦ id4[3]=xz
↦ id4[4]=y2
↦ id4[5]=xy
↦ id4[6]=x2
ideal id5=quot(id1,id2,2);
↦ [1023:2]4(14)s(13)s(12)s(11)s(10)s(9)s(8)s(7)s(6)s(5)s(4)s(3)s(2)sss
↦ product criterion:0 chain criterion:0
↦ (S:3)rtrtrt
↦ {1}[1023:3]4(2)sss
↦ product criterion:0 chain criterion:12
↦ {1}[1023:3]4(5)s(4)s(3)s(2)sss
↦ product criterion:0 chain criterion:9
↦ [255:3]2(8)s(7)s(6)s3(5)s(4)s(3)s(2)sss
↦ product criterion:15 chain criterion:0
id5;
↦ id5[1]=z2
↦ id5[2]=yz
↦ id5[3]=xz
↦ id5[4]=y2
↦ id5[5]=xy
↦ id5[6]=x2
```

See also: Section 5.1.103 [quotient], page 189.

5.1.102 quote

Syntax: `quote` (expression)

Type: none

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

Example:

```
link l="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);
↦ [1023:1]1(12)s2(11)s3(10)--s(7)s(6)-----7-
↦ product criterion:4 chain criterion:0
↦ _[1]=z
↦ _[2]=x2
↦ _[3]=xy2
↦ _[4]=y3
  close(l);
```

See Section 4.6.5.1 [MPfile links], page 86; Section 5.1.23 [eval], page 137; Section 5.1.137 [write], page 213.

5.1.103 quotient

Syntax: `quotient` (ideal_expression, ideal_expression)
`quotient` (module_expression, module_expression)

Type: ideal

Syntax: `quotient` (module_expression, ideal_expression)

Type: module

Purpose: computes the ideal quotient, resp. module quotient. Let `R` be the basering, `I,J` ideals and `M` a module in `R`^{n}. Then

`quotient(I,J)`$= \{a \in R \mid aJ \subset I\}$,

`quotient(M,J)`$= \{b \in R^n \mid bJ \subset M\}$.

Example:

```
ring r=181,(x,y,z),(c,ls);
ideal id1=maxideal(3);
ideal id2=x2+xyz,y2-z3y,z3+y5xz;
ideal id6=quotient(id1,id2);
id6;
↦ id6[1]=z
↦ id6[2]=y
↦ id6[3]=x
quotient(id2,id1);
↦ _[1]=z2
↦ _[2]=yz
↦ _[3]=y2
↦ _[4]=xz
↦ _[5]=xy
↦ _[6]=x2
module m=x*freemodule(3),y*freemodule(2);
ideal id3=x,y;
quotient(m,id3);
↦ _[1]=[1]
```

```
↦ _[2]=[0,1]
↦ _[3]=[0,0,x]
```

See Section 5.1.33 [fglmquot], page 142; Section 4.2 [ideal], page 68; Section 4.10 [module], page 100.

5.1.104 random

Syntax: `random` (int_expression, int_expression)

Type: int

Purpose: returns a random integer between the integer given by the first int_expression and the one given by the second int_expression.

Syntax: `random` (int_expression, int_expression, int_expression)

Type: intmat

Purpose: returns a random intmat where the size is given by the second (number of rows) and third argument (number of columns). The absolute value of the entries of the matrix is smaller than or equal to the integer given as the first argument.

Note: The random generator can be set to a start value with the function `system`, resp. by a command line option.

Example:

```
  random(1,1000);
↦ 35
  random(1,2,3);
↦ 0,0,0,
↦ 1,1,-1
  system("random",210);  // start random generator with 210
  random(-1000,1000);
↦ 707
  random(-1000,1000);
↦ 284
  system("random",210);
  random(-1000,1000);    // the same random values again
↦ 707
```

See Section 3.1.6 [Command line options], page 18; Section 4.3 [int], page 72; Section 4.4 [intmat], page 78; Section 5.1.124 [system], page 205.

5.1.105 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.
For MP links, one expression is read from the link and returned after evaluation.

For MPtcp links the `read` command blocks as long as there is no data to be read from the link. The `status` command can be used to check whether or not there is data to be read.
For DBM links, a `read` with one argument returns the value of the next entry in the data base, and a `read` with two arguments returns the value to the key given as the second argument from the data base.

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 5.1.26 [execute], page 138; Section 5.1.41 [getdump], page 146; Section 4.6 [link], page 83; Section 5.1.119 [status], page 201; Section 5.1.137 [write], page 213.

5.1.106 reduce

Syntax: `reduce` (poly_expression, ideal_expression)
`reduce` (poly_expression, ideal_expression, int_expression)
`reduce` (poly_expression, ideal_expression, poly_expression)
`reduce` (vector_expression, ideal_expression)
`reduce` (vector_expression, ideal_expression, int_expression)
`reduce` (vector_expression, module_expression)
`reduce` (vector_expression, module_expression, int_expression)
`reduce` (vector_expression, module_expression, poly_expression)
`reduce` (ideal_expression, ideal_expression)
`reduce` (ideal_expression, ideal_expression, int_expression)
`reduce` (ideal_expression, ideal_expression, matrix_expression)
`reduce` (module_expression, ideal_expression)
`reduce` (module_expression, ideal_expression, int_expression)
`reduce` (module_expression, module_expression)
`reduce` (module_expression, module_expression, int_expression)
`reduce` (module_expression, module_expression, matrix_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. Returns 0 if and only if the polynomial (resp. vector, ideal, module) is an element (resp. subideal, submodule) of the ideal (resp. module). The result may have no meaning if the second argument is not a standard basis.
The third (optional) argument 1 of type int forces a reduction which considers only the leading term and does no tail reduction.
If a third argument `u` of type poly or matrix is given, the first argument `p` is replaced by

p/u. This works only for zero dimensional ideals (resp. modules) in the second argument and gives, even in a local ring, a reduced normal form which is the projection to the quotient by the ideal (resp. module).

Note: The commands `reduce` and `NF` are synonymous.

Example:

```
  ring r1 = 0,(z,y,x),ds;
  poly s1=2x5y+7x2y4+3x2yz3;
  poly s2=1x2y2z2+3z8;
  poly s3=4xy5+2x2y2z3+11x10;
  ideal i=s1,s2,s3;
  ideal j=std(i);
  reduce(3z3yx2+7y4x2+yx5+z12y2x2,j);
↦ -yx5+2401/81y14x2+2744/81y11x5+392/27y8x8+224/81y5x11+16/81y2x14
  reduce(3z3yx2+7y4x2+yx5+z12y2x2,j,1);
↦ -yx5+z12y2x2
```

See Section 4.2 [ideal], page 68; Section 4.10 [module], page 100; Section 5.1.120 [std], page 202; Section 4.18 [vector], page 118.

5.1.107 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 $0 \rightarrow \bigoplus_a K[x]e_{a,n} \rightarrow \ldots \rightarrow \bigoplus_a K[x]e_{a,0} \rightarrow I \rightarrow 0$ be a minimal resolution of I considered with homogeneous maps of degree 0. The regularity is the smallest number s with the property $\deg(e_{a,i}) \leq s+i$ for all i.

Note: If applied to a non minimal resolution only an upper bound is returned.
If the input to the commands `res` and `mres` is homogeneous the regularity is computed and used as a degree bound during the computation unless `option(notRegularity);` is given.

Example:

```
  ring rh3=32003,(w,x,y,z),(dp,C);
  poly f=x11+y10+z9+x5y2+x2y2z3+xy3*(y2+x)^2;
  ideal j=homog(jacob(f),w);
  def jr=res(j,0);
  regularity(jr);
↦ 25
  // example for upper bound behavior:
  list jj=jr;
  regularity(jj);
↦ 25
  jj=nres(j,0);
  regularity(jj);
↦ 27
  jj=minres(jj);
```

See Section 4.7 [list], page 91; Section 5.1.74 [minres], page 168; Section 5.1.78 [mres], page 171; Section 5.1.89 [option], page 177; Section 5.1.109 [res], page 193; Section 4.15 [resolution], page 112; Section 5.1.118 [sres], page 200.

5.1.108 repart

Syntax: `repart` (number_expression)

Type: number

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

Example:

```
  ring r=(complex,i),x,dp;
  repart(1+2*i);
↦ 1
```

See Section 5.1.49 [impart], page 152.

5.1.109 res

Procedure from library `standard.lib` (see Section D.1 [standard_lib], page 293).

Syntax: `res` (ideal_expression`,` int_expression `[,` any_expression `])`
`res` (module_expression`,` int_expression `[,` any_expression `])`

Type: resolution

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

Depending on the input, the returned resolution is computed using the following methods:

quotient rings:
`nres` (classical method using syzygies) , see Section 5.1.85 [nres], page 175.

homogeneous ideals and k=0:
`lres` (La'Scala's method), see Section 5.1.69 [lres], page 165.

not minimized resolution and (homogeneous input with k not 0, or local rings):
`sres` (Schreyer's method), see Section 5.1.118 [sres], page 200.

all other inputs:
`mres` (classical method), see Section 5.1.78 [mres], page 171.

Note: Accessing single elements of a resolution may require that some partial computations have to be finished and may therefore take some time.

See also Section 5.1.3 [betti], page 124; Section 5.1.47 [hres], page 151; Section 4.2 [ideal], page 68; Section 5.1.69 [lres], page 165; Section 5.1.74 [minres], page 168; Section 4.10 [module], page 100; Section 5.1.78 [mres], page 171; Section 5.1.85 [nres], page 175; Section 4.15 [resolution], page 112; Section 5.1.118 [sres], page 200.

Example:

```
  ring r=0,(x,y,z),dp;
ideal i=xz,yz,x3-y3;
def l=res(i,0); // homogeneous ideal: uses lres
l;
↦  1      3      2
↦ r <--  r <--  r
↦
↦ 0      1      2
↦ resolution not minimized yet
↦
print(betti(l), "betti"); // input to betti may be of type resolution
↦            0     1     2
↦ ------------------------
↦     0:     1     -     -
↦     1:     -     2     1
↦     2:     -     1     1
↦ ------------------------
↦ total:     1     3     2
l[2];         // element access may take some time
↦ _[1]=-x*gen(1)+y*gen(2)
↦ _[2]=-x2*gen(2)+y2*gen(1)+z*gen(3)
i=i,x+1;
l=res(i,0);   // inhomogeneous ideal: uses mres
l;
↦  1      3      3      1
↦ r <--  r <--  r <--  r
↦
↦ 0      1      2      3
↦ resolution not minimized yet
↦
ring rs=0,(x,y,z),ds;
ideal i=imap(r,i);
def l=res(i,0); // local ring not minimized: uses sres
l;
↦   1       1
↦ rs <--  rs
↦
↦ 0       1
↦ resolution not minimized yet
↦
res(i,0,0);     // local ring and minimized: uses mres
↦   1       1
↦ rs <--  rs
↦
↦ 0       1
↦
```

5.1.110 reservedName

Syntax: `reservedName ()`

Type: none

Syntax: `reservedName (` string_expression `)`

Type: int

Purpose: prints a list of all reserved identifiers (first form) or tests whether the string is a reserved identifier (second form).

Example:

```
reservedName();
↦ ... // output skipped
  reservedName("ring");
↦ 1
  reservedName("xyz");
↦ 0
```

See Section 5.1.82 [names], page 173; Section 4.17 [string], page 114.

5.1.111 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 107; Section 4.16 [ring], page 113.

5.1.112 rvar

Syntax: `rvar` (name)
`rvar` (poly_expression)
`rvar` (string_expression)

Type: int

Purpose: returns the number of the variable if the name is a ring variable of the basering or if the string is the name of a ring variable of the basering; returns 0 if not. Hence the return value of `rvar` can also be used in a boolean context to check whether the variable exists.

Example:

```
  ring r=29,(x,y,z),lp;
  rvar(x);
↦ 1
  rvar(r);
↦ 0
  rvar(y);
↦ 2
  rvar(var(3));
↦ 3
  rvar("x");
↦ 1
```

See Section 5.1.13 [defined], page 131; Section 4.16 [ring], page 113; Section 5.1.132 [var], page 211; Section 5.1.133 [varstr], page 211.

5.1.113 setring

Syntax: `setring` ring_name

Type: none

Purpose: changes the basering to another (already defined) ring.

Example:

```
  ring r1=0,(x,y),lp;
  // the basering is r1
  ring r2=32003,(a(1..8)),ds;
  // the basering is r2
  setring r1;
  // the basering is again r1
  nameof(basering);
↦ r1
  listvar();
↦ // r2                  [0]  ring
↦ // r1                  [0]  *ring
↦ // LIB                 [0]  string standard.lib
```

Use in procedures:
All changes of the basering by a definition of a new ring or a `setring` command in a procedure are local to this procedure. Use `keepring` to move a ring, which is local to a procedure, up by one nesting level.

See Section 5.2.8 [keepring], page 217; Section 4.14 [qring], page 111; Section 4.16 [ring], page 113.

5.1.114 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 simplification rules are the sum of the following functions:

`1` normalize (make leading coefficients 1).

`2` erase zero generators/columns.

`4` keep only the first one of identical generators/columns.

`8` keep only the first one of generators/columns which differ only by a factor in the ground field.

`16` keep only those generators/columns whose leading monomials differ.

`32` keep only those generators/columns whose leading monomials are not divisible by other ones.

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 4.2 [ideal], page 68; Section 4.10 [module], page 100; Section 4.12 [poly], page 107; Section 4.18 [vector], page 118.

5.1.115 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_expression)
`size` (ring_expression)

Type: int

Purpose: depends on the type of argument:

ideal or module
returns the number of (non-zero) generators.

string, intvec, list or resolution
returns the length, i.e., the number of characters, entries or elements.

poly or vector
returns the number of monomials.

matrix or intmat
returns the number of entries (rows*columns).

ring returns the number of elements in the ground field (for Z/p and algebraic extensions) or -1

Example:

```
  string s="hello";
  size(s);
↦ 5
  intvec iv=1,2;
  size(iv);
↦ 2
  ring r=0,(x,y,z),lp;
  poly f=x+y+z;
  size(f);
↦ 3
  vector v=[x+y,0,0,1];
  size(v);
↦ 3
  ideal i=f,y;
  size(i);
↦ 2
  module m=v,[0,1],[0,0,1],2*v;
  size(m);
↦ 4
  matrix mm[2][2];
  size(mm);
↦ 4
  ring r1=(2,a),x,dp;
  minpoly=a4+a+1;
  size(r1);
↦ 8
```

See Section 4.2 [ideal], page 68; Section 4.4 [intmat], page 78; Section 4.5 [intvec], page 81; Section 4.10 [module], page 100; Section 5.1.83 [ncols], page 174; Section 5.1.86 [nrows], page 176; Section 4.12 [poly], page 107; Section 4.17 [string], page 114; Section 4.18 [vector], page 118.

5.1.116 sortvec

Syntax: `sortvec` (ideal_expression)
`sortvec` (module_expression)

Type: intvec

Purpose: computes the permutation `v` which orders the ideal, resp. module, `I` by its initial terms, starting with the smallest, that is, `I(v[i]) < I(v[i+1])` for all `i`.

Example:

```
  ring r=0,(x,y,z),dp;
  ideal I=y,z,x,x3,xz;
  sortvec(I);
↦ 2,1,3,5,4
```

See Section D.2.2 [general_lib], page 294.

5.1.117 sprintf

Procedure from library `standard.lib` (see Section D.1 [standard_lib], page 293).

Syntax: `sprintf` (string_expression `[,` any_expressions `]`)

Return: string

Purpose: `sprintf(fmt,...);` performs output formatting. The first argument is a format control string. Additional arguments may be required, depending on the content of the control string. A series of output characters is generated as directed by the control string; these characters are returned as a string.
The control string `fmt` is simply text to be copied, except that the string may contain conversion specifications.
Do `help print;` for a listing of valid conversion specifications. As an addition to the conversions of `print`, the `%n` and `%2` conversion specification does not consume an additional argument, but simply generates a newline character.

Note: If one of the additional arguments is a list, then it should be enclosed once more into a `list()` command, since passing a list as an argument flattens the list by one level.

Example:

```
  ring r=0,(x,y,z),dp;
module m=[1,y],[0,x+z];
intmat M=betti(mres(m,0));
list l = r, m, M;
string s = sprintf("s:%s,%n l:%l", 1, 2); s;
↦ s:1,
↦  l:int(2)
s = sprintf("s:%n%s", l); s;
↦ s:
↦ (0),(x,y,z),(dp(3),C)
s = sprintf("s:%2%s", list(l)); s;
↦ s:
↦ (0),(x,y,z),(dp(3),C),y*gen(2)+gen(1),x*gen(2)+z*gen(2),1,1
s = sprintf("2l:%n%2l", list(l)); s;
↦ 2l:
↦ list("(0),(x,y,z),(dp(3),C)",
↦ module(y*gen(2)+gen(1),
↦ x*gen(2)+z*gen(2)),
```

```
↦ intmat(intvec(1,1 ),1,2))
↦
s = sprintf("%p", list(l)); s;
↦ [1]:
↦    //   characteristic : 0
↦ //   number of vars : 3
↦ //        block   1 : ordering dp
↦ //                  : names    x y z
↦ //        block   2 : ordering C
↦ [2]:
↦    _[1]=y*gen(2)+gen(1)
↦    _[2]=x*gen(2)+z*gen(2)
↦ [3]:
↦    1,1
↦
s = sprintf("%;", list(l)); s;
↦ [1]:
↦    //   characteristic : 0
↦ //   number of vars : 3
↦ //        block   1 : ordering dp
↦ //                  : names    x y z
↦ //        block   2 : ordering C
↦ [2]:
↦    _[1]=y*gen(2)+gen(1)
↦    _[2]=x*gen(2)+z*gen(2)
↦ [3]:
↦    1,1
↦
s = sprintf("%b", M); s;
↦             0     1
↦ ------------------
↦     0:     1     1
↦ ------------------
↦ total:     1     1
↦
```

See also: Section 5.1.37 [fprintf], page 144; Section 5.1.97 [print], page 183; Section 5.1.98 [printf], page 185; Section 4.17 [string], page 114.

5.1.118 sres

Syntax: `sres` (ideal_expression`,` int_expression)
`sres` (module_expression`,` int_expression)

Type: resolution

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

$$... \longrightarrow F_2 \xrightarrow{A_2} F_1 \xrightarrow{A_1} F_0 \longrightarrow F_0/M \longrightarrow 0.$$

If the int expression k is not zero then the computation stops after k steps and returns a list of modules (given by standard bases) M_i = `module`(A_i), i=1..k.
`sres(M,0)` returns a list of n modules where n is the number of variables of the basering.

Even if `sres` does not compute a minimal resolution, the `betti` command gives the true betti numbers! In many cases of interest `sres` is much faster than any other known method. Let `list L=sres(M,0);` then `L[1]=M` is identical to the input, `L[2]` is a standard basis with respect to the Schreyer ordering of the first syzygy module of `L[1]`, etc. (`L[i]` $= M_i$ in the notations from above.)

Note: Accessing single elements of a resolution may require that some partial computations have to be finished and may therefore take some time.

Example:

```
  ring r=31991,(t,x,y,z,w),ls;
  ideal M=t2x2+tx2y+x2yz,t2y2+ty2z+y2zw,
          t2z2+tz2w+xz2w,t2w2+txw2+xyw2;
  M=std(M);
  resolution L=sres(M,0);
  L;
↦  1      35      141      209      141      43      4
↦ r <--  r <--   r <--    r <--    r <--    r <--   r
↦
↦ 0      1       2        3        4        5       6
↦ resolution not minimized yet
↦
  print(betti(L),"betti");
↦            0     1     2     3     4     5
↦ ------------------------------------------
↦     0:     1     -     -     -     -     -
↦     1:     -     -     -     -     -     -
↦     2:     -     -     -     -     -     -
↦     3:     -     4     -     -     -     -
↦     4:     -     -     -     -     -     -
↦     5:     -     -     -     -     -     -
↦     6:     -     -     6     -     -     -
↦     7:     -     -     9    16     2     -
↦     8:     -     -     -     2     5     1
↦ ------------------------------------------
↦ total:     1     4    15    18     7     1
```

See Section 5.1.3 [betti], page 124; Section 5.1.47 [hres], page 151; Section 4.2 [ideal], page 68; Section 4.3 [int], page 72; Section 5.1.69 [lres], page 165; Section 5.1.74 [minres], page 168; Section 4.10 [module], page 100; Section 5.1.78 [mres], page 171; Section 5.1.109 [res], page 193; Section 5.1.125 [syz], page 207.

5.1.119 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.28 [Parallelization with MPtcp links], page 272 for an example). Note that on some systems, the minimum time for a process to be put to sleep is one second.
The following string expressions are allowed:

`"name"` the name string given by the definition of the link (usually the filename)

`"type"` returns `"ASCII"`, `"MPfile"`, `"MPtcp"` or `"DBM"`

`"open"` returns `"yes"` or `"no"`

`"openread"`
returns `"yes"` or `"no"`

`"openwrite"`
returns `"yes"` or `"no"`

`"read"` returns `"ready"` or `"not ready"`

`"write"` returns `"ready"` or `"not ready"`

`"mode"` 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 83; Section 5.1.88 [open], page 177; Section 5.1.105 [read], page 190; Section 5.1.137 [write], page 213.

5.1.120 std

Syntax: `std (` ideal_expression)
`std (` module_expression)
`std (` ideal_expression`,` intvec_expression `)`
`std (` module_expression`,` intvec_expression `)`
`std (` ideal_expression`,` intvec_expression`,` intvec_expression `)`
`std (` module_expression`,` intvec_expression`,` intvec_expression `)`
`std (` ideal_expression`,` poly_expression `)`
`std (` module_expression`,` vector_expression `)`

Type: ideal or module

Purpose: returns a standard basis of an ideal or module with respect to the monomial ordering of the basering. A standard basis is a set of generators such that the leading terms generate the leading ideal, resp. module.
Use an optional second argument of type intvec as Hilbert series (result of `hilb(i,1)`, see Section 5.1.45 [hilb], page 149), if the ideal, resp. module, is homogeneous (Hilbert

driven standard basis computation, Section 5.1.122 [stdhilb], page 204). If the ideal is quasihomogeneous with some weights w and if the Hilbert series is computed w.r.t. to these weights, then use w as third argument.
Use an optional second argument of type poly, resp. vector, to construct the standard basis from an already computed one (given as the first argument) and one additional generator (the second argument).

Note: The standard basis is computed with a (more or less) straight-forward implementation of the classical Buchberger (resp. Mora) algorithm. For global orderings, use the `groebner` command instead (see Section 5.1.42 [groebner], page 147), which heuristically chooses the "best" algorithm to compute a Groebner basis.
To view the progress of long running computations, use `option(prot)` (see [option(prot)], page 178).

Example:

```
  // local computation
  ring r=32003,(x,y,z),ds;
  poly s1=1x2y+151xyz10+169y21;
  poly s2=1xz14+6x2y4+3z24;
  poly s3=5y10z10x+2y20z10+y10z20+11x3;
  ideal i=s1,s2,s3;
  ideal j=std(i);
  degree(j);
↦ 0
  // Hilbert driven elimination (standard)
  ring rhom=32003,(x,y,z,h),dp;
  ideal i=homog(imap(r,i),h);
  ideal j=std(i);
  intvec iv=hilb(j,1);
  ring rlex=32003,(x,y,z,h),lp;
  ideal i=fetch(rhom,i);
  ideal j=std(i,iv);
  j=subst(j,h,1);
  j[1];
↦ z64
  // Hilbert driven elimination (ideal is quasihomogeneous)
  intvec w=10,1,1;
  ring whom=32003,(x,y,z),wp(w);
  ideal i=fetch(r,i);
  ideal j=std(i);
  intvec iw=hilb(j,1,w);
  ring wlex=32003,(x,y,z),lp;
  ideal i=fetch(whom,i);
  ideal j=std(i,iw,w);
  j[1];
↦ z64
```

See Section 5.1.29 [facstd], page 139; Section 5.1.32 [fglm], page 142; Section 5.1.42 [groebner], page 147; Section 4.2 [ideal], page 68; Section 5.1.79 [mstd], page 171; Section 5.1.89 [option], page 177; Section 4.16 [ring], page 113; Section 5.1.121 [stdfglm], page 204; Section 5.1.122 [stdhilb], page 204.

5.1.121 stdfglm

Procedure from library `standard.lib` (see Section D.1 [standard_lib], page 293).

Syntax: `stdfglm` (ideal_expression)
`stdfglm` (ideal_expression, string_expression)

Type: ideal

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

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 also: Section 5.1.32 [fglm], page 142; Section 5.1.42 [groebner], page 147; Section 5.1.120 [std], page 202; Section 5.1.122 [stdhilb], page 204.

5.1.122 stdhilb

Procedure from library `standard.lib` (see Section D.1 [standard_lib], page 293).

Syntax: `stdhilb` (ideal_expression)
`stdhilb` (ideal_expression, intvec_expression)

Type: ideal

Purpose: computes the standard basis of the homogeneous ideal in the basering, via a Hilbert driven standard basis computation.
An optional second argument will be used as 1st Hilbert function.

Assume: The optional second argument is the first Hilbert series as computed by `hilb`.

Example:

```
  ring  r=0,(x,y,z),dp;
ideal i=y3+x2,x2y+x2,x3-x2,z4-x2-y;
ideal i1=stdhilb(i); i1;
↦ i1[1]=y3+x2
↦ i1[2]=x2y+x2
↦ i1[3]=x3-x2
↦ i1[4]=z4-x2-y
```

```
// the latter computation is equivalent to:
intvec v=hilb(i,1);
↦ // ** i is no standardbasis
ideal i2=stdhilb(i,v); i2;
↦ i2[1]=y3+x2
↦ i2[2]=x2y+x2
↦ i2[3]=x3-x2
↦ i2[4]=z4-x2-y
```

See also: Section 5.1.42 [groebner], page 147; Section 5.1.120 [std], page 202; Section 5.1.121 [stdfglm], page 204.

5.1.123 subst

Syntax: `subst` (poly_expression, ring_variable, poly_expression)
`subst` (vector_expression, ring_variable, poly_expression)
`subst` (ideal_expression, ring_variable, poly_expression)
`subst` (module_expression, ring_variable, poly_expression)

Type: poly, vector, ideal or module (corresponding to the first argument)

Purpose: substitutes a ring variable by a polynomial.

Example:

```
  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
  subst(f,y,z+1);
↦ x2+2z2+x+4z+2
```

See Section 4.2 [ideal], page 68; Section 4.8 [map], page 94; Section 4.10 [module], page 100; Section 4.12 [poly], page 107; Section 4.18 [vector], page 118.

5.1.124 system

Syntax: `system` (string_expression)
`system` (string_expression, expression)

Type: depends on the desired function, may be none

Purpose: interface to internal data and the operating system. The string_expression determines the command to execute. Some commands require an additional argument (second form) where the type of the argument depends on the command. See below for a list of all possible commands.

Note: Not all functions work on every platform.

Functions:

`system("sh"`, string_expression)
: shell escape, returns the return code of the shell as int. The string is sent literally to the shell.

`system("pid")`
: returns the process number as int (for creating unique names).

`system("uname")`
: returns a string identifying the architecture for which SINGULAR was compiled.

`system("getenv",` string_expression)
: returns the value of the shell environment variable given as the second argument. The return type is string.

`system("setenv",`string_expression, string_expression)
: sets the shell environment variable given as the second argument to the value given as the third argument. Returns the third argument. Might not be available on all platforms.

`system("tty")`
: resets the terminal.

`system("version")`
: returns the version number of SINGULAR as int.

`system("contributors")`
: returns names of people who contributed to the SINGULAR kernel as string.

`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")`

`system("nblocks",` ring_name)
: returns the number of blocks of the given ring, or the number of parameters of the current basering, if no second argument is given. The return type is int.

`system("Singular")`
: returns the absolute (path) name of the running SINGULAR as string.

`system("–")`
: prints the values of all options.

`system("`–long_option_name`")`
: returns the value of the (command-line) option long_option_name. The type of the returned value is either string or int. See Section 3.1.6 [Command line options], page 18, for more info.

`system("`–long_option_name`",` expression)
: sets the value of the (command-line) option long_option_name to the value given by the expression. Type of the expression must be string, or int. See Section 3.1.6 [Command line options], page 18, for more info. Among others, this can be used for setting the seed of the random number generator, the used help browser, the minimal display time, or the timer resolution.

`system("browsers");`
: returns a string about available help browsers. See Section 3.1.3 [The online help system], page 14.

Example:

```
// a listing of the current directory:
system("sh","ls");
// execute a shell, return to SINGULAR with exit:
system("sh","sh");
string unique_name="/tmp/xx"+string(system("pid"));
unique_name;
↦ /tmp/xx4711
system("uname")
↦ ix86-Linux
system("getenv","PATH");
↦ /bin:/usr/bin:/usr/local/bin
system("Singular");
↦ /usr/local/bin/Singular
// report value of all options
system("--");
↦ // --batch           0
↦ // --execute
↦ // --sdb             0
↦ // --echo            0
↦ // --help            0
↦ // --quiet           0
↦ // --random          937848971
↦ // --no-tty          0
↦ // --user-option
↦ // --version         0
↦ // --allow-net       0
↦ // --browser
↦ // --emacs           0
↦ // --no-stdlib       0
↦ // --no-rc           0
↦ // --no-warn         0
↦ // --no-out          0
↦ // --min-time        "0.5"
↦ // --mp-port
↦ // --mp-host
↦ // --ticks-per-sec   1
↦ // --mp-transp
↦ // --mp-mode
// set minimal display time to 0.02 seconds
system("--min-time", "0.02");
// set timer resolution to 0.01 seconds
system("--ticks-per-sec", 100);
// re-seed random number generator
system("--random", 12345678);
// allow netscape to access HTML pages from the net
system("--allow-net", 1);
// and set help browser to netscape
system("--browser", "netscape");
```

5.1.125 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 5.1.47 [hres], page 151; Section 4.2 [ideal], page 68; Section 5.1.69 [lres], page 165; Section 4.10 [module], page 100; Section 5.1.78 [mres], page 171; Section 5.1.85 [nres], page 175; Section 5.1.109 [res], page 193; Section 5.1.118 [sres], page 200.

5.1.126 trace

Syntax: `trace` (intmat_expression)
`trace` (matrix_expression)

Type: int, if the argument is an intmat, resp.
poly, if the argument is a matrix

Purpose: returns the trace of an intmat, resp. matrix.

Example:

```
  intmat m[2][2]=1,2,3,4;
  print(m);
↦      1     2
↦      3     4
  trace(m);
↦ 5
```

See Section 4.4 [intmat], page 78; Section 4.9 [matrix], page 96.

5.1.127 transpose

Syntax: `transpose` (intmat_expression)
`transpose` (matrix_expression)
`transpose` (module_expression)

Type: intmat, matrix, or module, corresponding to the argument

Purpose: transposes a matrix.

Example:

```
  ring R=0,x,dp;
  matrix m[2][3]=1,2,3,4,5,6;
  print(m);
↦ 1,2,3,
↦ 4,5,6
  print(transpose(m));
↦ 1,4,
↦ 2,5,
↦ 3,6
```

See Section 4.4 [intmat], page 78; Section 4.9 [matrix], page 96; Section 4.10 [module], page 100.

5.1.128 type

Syntax: `type` name ;
`type (` name `);`

Type: none

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

Example:

```
  int i=3;
  i;
↦ 3
  type(i);
↦ // i                    [0]  int 3
```

See Chapter 4 [Data types], page 67; Section 5.1.68 [listvar], page 164; Section 5.1.97 [print], page 183.

5.1.129 typeof

Syntax: `typeof (` expression `)`

Type: string

Purpose: returns the type of an expression as string.

Returns the type of the first list element if the expression is an expression list.

Possible types are: `"ideal"`, `"int"`, `"intmat"`, `"intvec"`, `"list"`, `"map"`, `"matrix"`, `"module"`, `"number"`, `"none"`, `"poly"`, `"proc"`, `"qring"`, `"resolution"`, `"ring"`, `"string"`, `"vector"`.

For internal use only is the type `"?unknown type?"`.

Example:

```
  int i=9; i;
↦ 9
  typeof(_);
↦ int
  print(i);
↦ 9
  typeof(_);
↦ 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 67; Section 5.1.128 [type], page 209.

5.1.130 uressolve

Syntax: `uressolve` (ideal_expression, int_expression, int_expression, int_expression)

Type: list

Purpose: computes all complex roots of a zerodimensional ideal.
Makes either use of the multipolynomial resultant of Macaulay (second argument = 1), which works only for homogeneous ideals, or uses the sparse resultant of Gelfand, Kapranov and Zelevinsky (second argument = 0).
The sparse resultant algorithm uses a mixed polyhedral subdivision of the Minkowsky sum of the Newton polytopes in order to construct the sparse resultant matrix. Its determinant is a nonzero multiple of the sparse resultant. The u-resultant of B.L. van der Waerden and Laguerre's algorithm are used to determine the complex roots.
The third argument defines the precision of the fractional part if the ground field is the field of rational numbers, otherwise it will be ignored.
The fourth argument (can be 0, 1 or 2) gives the number of extra runs of Laguerre's algorithm (with corrupted roots), leading to better results.

Note: If the ground field is the field of complex numbers, the elements of the list are of type number, otherwise of type string.

Example:

```
ring rsc=(real,20,I),(x,y),lp;
ideal i=(2+3*I)*x2+(0.35+I*45.0e-2)*y2-8,x2+xy+(42.7)*y2;
list l=uressolve(i,0,0,2);
l[1];
↦ [1]:
↦    (-1.315392899374542198+I*0.70468233142752928117)
↦ [2]:
↦    (0.12292646536251281054+I*0.19245727404407015049)
subst(subst(i[1],x,l[1][1]),y,l[1][2]);
↦ 0
```

See Section 5.1.60 [laguerre], page 159; Section 5.1.77 [mpresmat], page 170.

5.1.131 vandermonde

Syntax: `vandermonde` (ideal_expression, ideal_expression, int_expression)

Type: poly

Purpose: `vandermonde(p,v,d)` computes the (unique) polynomial of degree `d` with prescribed values `v[1],...,v[N]` at the points $\mathtt{p}^0,\ldots,\mathtt{p}^{N-1}$, `N=(d+1)`^{n}, n the number of ring variables.

The returned polynomial is $\sum c_{\alpha_1\ldots\alpha_n}\cdot x_1^{\alpha_1}\cdot\ldots\cdot x_n^{\alpha_n}$, where the coefficients $c_{\alpha_1\ldots\alpha_n}$ are the solution of the (transposed) Vandermonde system of linear equations

$$\sum_{\alpha_1+\ldots+\alpha_n\le d} c_{\alpha_1\ldots\alpha_n}\cdot\mathtt{p}_1^{(k-1)\alpha_1}\cdot\ldots\cdot\mathtt{p}_n^{(k-1)\alpha_n}=\mathtt{v}[k],\quad k=1,\ldots,\mathtt{N}.$$

Note: the ground field has to be the field of rational numbers. Moreover, `ncols(p)==`n, the number of variables in the basering, and all the given generators have to be numbers different from 0,1 or -1. Finally, `ncols(v)==(d+1)`^{n}, and all given generators have to be numbers.

Example:

```
ring r=0,(x,y),dp;
// determine f with deg(f)=2 and with given values v of f
// at 9 points: (2,3)^0=(1,1),...,(2,3)^8=(2^8,3^8)
// valuation point: (2,3)
ideal p=2,3;
ideal v=1,2,3,4,5,6,7,8,9;
poly ip=vandermonde(p,v,2);
ip[1..5];  //  the 5 first terms of ip:
↦ -1/9797760x2y2-595/85536x2y+55/396576xy2+935/384x2-1309/3240xy
// compute value of ip at the point 2^8,3^8, result must be 9
subst(subst(ip,x,2^8),y,3^8);
↦ 9
```

5.1.132 var

Syntax: `var` (int_expression)

Type: poly

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

Example:

```
  ring r=0,(x,y,z),dp;
  var(2);
↦ y
```

See Section 4.3 [int], page 72; Section 5.1.87 [nvars], page 176; Section 4.16 [ring], page 113; Section 5.1.133 [varstr], page 211.

5.1.133 varstr

Syntax: `varstr` (ring_name)
`varstr` (int_expression)
`varstr` (ring_name`,` int_expression)

Type: string

Purpose: returns the list of the names of the ring variables as a string or the name of the n-th ring variable, where n is given by the int_expression.
If the ring name is omitted, the basering is used, thus `varstr(n)` is equivalent to `varstr(basering,n)`.

Example:

```
  ring r=0,(x,y,z),dp;
  varstr(r);
↦ x,y,z
  varstr(r,1);
↦ x
  varstr(2);
↦ y
```

See Section 5.1.6 [charstr], page 126; Section 4.3 [int], page 72; Section 5.1.87 [nvars], page 176; Section 5.1.91 [ordstr], page 181; Section 5.1.94 [parstr], page 182; Section 4.16 [ring], page 113; Section 5.1.132 [var], page 211.

5.1.134 vdim

Syntax: `vdim` (ideal_expression)
`vdim` (module_expression)

Type: int

Purpose: computes the vector space dimension of the ring, resp. free module, modulo the ideal, resp. module, generated by the initial terms of the given generators. If the generators form a standard basis, this is the same as the vector space dimension of the ring, resp. free module, modulo the ideal, resp. module.
If the ideal, resp. module, is not zero-dimensional, -1 is returned.

Example:

```
  ring r=0,(x,y),ds;
  ideal i=x2+y2,x2-y2;
  ideal j=std(i);
  vdim(j);
↦ 4
```

See Section 5.1.15 [degree], page 132; Section 5.1.19 [dim], page 134; Section 4.2 [ideal], page 68; Section 5.1.56 [kbase], page 157; Section 5.1.80 [mult], page 172; Section 5.1.120 [std], page 202.

5.1.135 wedge

Syntax: `wedge` (matrix_expression, int_expression)

Type: matrix

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

Example:

```
  ring r;
  matrix m[2][3]=x,y,y,z,z,x;
  print(m);
↦ x,y,y,
↦ z,z,x
  print(wedge(m,2));
↦ xz-yz,-x2+yz,xy-yz
```

See Section 4.3 [int], page 72; Section 4.9 [matrix], page 96; Section 5.1.73 [minor], page 168.

5.1.136 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 68; Section 4.5 [intvec], page 81; Section 5.1.100 [qhweight], page 187.

5.1.137 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 values of the variables f and i as strings into
// the file "outfile" (overwrite it, if it exists)
write(":w outfile",f,i);

// now append the string "that was f,i" (without the quotes)
// at the end of the file "outfile"
write(":a outfile","that was f,i");
// alternatively, links could be used:
link l=":a outfile"; l;
// type : ASCII
// mode : a
// name : outfile
// open : no
// read : not ready
// write: not ready
write(l," that was f,i");
// saving and retrieving data (ASCII format):
```

```
ring r=32003,(x,y,z),dp;
ideal i=x+y,z3+22y;
write(":w save_i",i);// this writes x+y,z3+22y to the file save_i
ring r=32003,(x,y,z),dp;
string s=read("save_i");   //creates the string x+y,z3+22y
execute("ideal k="+s+";"); // this defines an ideal k which
                           // is equal to i.
// for large objects, the MP format and MPfile links are better:
write("MPfile:w save_i.mp",i);
def j=read("MPfile:r save_i.mp");
```

See Chapter 4 [Data types], page 67; Section 5.1.21 [dump], page 135; Section 5.1.23 [eval], page 137; Section 4.6 [link], page 83; Section 5.1.97 [print], page 183; Section 5.1.98 [printf], page 185; Section 5.1.102 [quote], page 188; Section 5.1.105 [read], page 190; Section 5.3.7 [short], page 222.

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;
  temp=i;
  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 214; Section 5.2.6 [for], page 216; Section 5.2.11 [while], page 219.

5.2.2 breakpoint

Syntax: `breakpoint(` proc_name `);`
`breakpoint(` proc_name, line_no `);`

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

Example:

```
breakpoint(groebner);
↦ breakpoint 1, at line 162 in groebner
breakpoint(groebner, 176);
↦ breakpoint 2, at line 176 in groebner
breakpoint(groebner, -1);
↦ breakpoints in groebner deleted(0x6)
```

See Section 3.10.2 [Source code debugger], page 64; Section 5.2.12 [~], page 219.

5.2.3 continue

Syntax: `continue;`

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

Note: Unlike the C-construct it **does not execute the increment statement**. The command `continue` is mainly for internal use.

Example:

```
for (int i = 1 ; i<=10; i=i+1)
{
   ...
   if (i==3) { i=8;continue; }
     // skip the rest if i is 3 and
     // continue with the next i: 8
   i;
}
↦ 1
↦ 2
↦ 8
↦ 9
↦ 10
```

See Section 5.2 [Control structures], page 214; Section 5.2.6 [for], page 216; Section 5.2.11 [while], page 219.

5.2.4 else

Syntax: `if (` boolean_expression `)` true_block `else` false_block

Purpose: executes false_block if the boolean_expression of the `if` statement is false. This command is only valid in combination with an `if` command.

Example:

```
int i=3;
if (i > 5)
{
  "i is bigger than 5";
}
else
{
  "i is smaller than 6";
}
↦ i is smaller than 6
```

See Section 5.2 [Control structures], page 214; Section 4.3.5 [boolean expressions], page 76; Section 5.2.7 [if], page 217.

5.2.5 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 5.2.8 [keepring], page 217.

5.2.6 for

Syntax: `for (` init_command`;` boolean_expression`;` iterate_commands`)` block

Purpose: repetitive, conditional execution of a command block.
The command init_command is executed first. Then boolean_expression is evaluated. If its value is TRUE the block is executed, otherwise the `for` statement is complete. After each execution of the block, the command iterate_command is executed and boolean_expression is evaluated. This is repeated until boolean_expression evaluates to FALSE.
The command `break;` leaves the innermost `for` construct.

Example:

```
// sum of 1 to 10:
int s=0;
for (int i=1; i<=10; i=i+1)
{
   s=s+i;
}
s;
↦ 55
```

See Section 5.2 [Control structures], page 214; Section 4.3.5 [boolean expressions], page 76; Section 5.2.1 [break], page 214; Section 5.2.3 [continue], page 215; Section 5.2.7 [if], page 217; Section 5.2.11 [while], page 219.

5.2.7 if

Syntax: `if` (boolean_expression) true_block
`if` (boolean_expression) true_block `else` false_block

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

Example:

```
int i = 9;
matrix m[i][i];
if (i > 5 and typeof(m) == "matrix")
{
  m[i][i] = i;
}
```

See Section 5.2 [Control structures], page 214; Section 4.3.5 [boolean expressions], page 76; Section 5.2.1 [break], page 214; Section 5.2.4 [else], page 215.

5.2.8 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
```

See Section 4.16 [ring], page 113.

5.2.9 quit

Syntax: `quit;`

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

Example:

```
quit;
```

5.2.10 return

Syntax: `return` (expression_list `);`
`return ();`

Type: any

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

Example:

```
proc p2
{
  int i,j;
  for(i=1;i<=10;i++)
  {
    j=j+i;
  }
  return(j);
}
// can also return an expression list, i.e., more than one value
proc tworeturn ()
{ return (1,2); }
int i,j = tworeturn();
// return type may even depend on the input
proc type_return (int i)
{
  if (i > 0) {return (i);}
  else {return (list(i));}
}
// then we need def type (or list) to collect value
def t1 = type_return(1);
def t2 = type_return(-1);
```

See Chapter 4 [Data types], page 67; Section 4.13 [proc], page 110.

5.2.11 while

Syntax: `while` (boolean_expression) block

Purpose: repetitive, conditional execution of block.
The boolean_expression is evaluated and if its value is TRUE, the block gets executed. This is repeated until boolean_expression evaluates to FALSE. The command `break` leaves the innermost `while` construction.

Example:

```
int i = 9;
while (i>0)
{
   // ... // do something for i=9, 8, ..., 1
   i = i - 1;
}
while (1)
{
   // ...   // do something forever
   if (i == -5) // but leave the loop if i is -5
   {
     break;
   }
}
```

See Section 5.2 [Control structures], page 214; Section 4.3.5 [boolean expressions], page 76; Section 5.2.1 [break], page 214.

5.2.12 ~ (break point)

Syntax: `~;`

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

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 3.10.3 [Break points], page 65.

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 131; Section 4.3 [int], page 72; Section 5.1.89 [option], page 177; Section 5.1.120 [std], page 202.

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 72; Section 5.3.11 [voice], page 224.

5.3.3 minpoly

Type: number

Purpose: describes the coefficient field of the current basering as an algebraic extension with the minimal polynomial equal to `minpoly`. Setting the `minpoly` should be the first command after defining the ring.

Note: The minimal polynomial has to be specified in the syntax of a polynomial. Its variable is not one of the ring variables, but the algebraic element which is being adjoined to the field. Algebraic extensions in SINGULAR are only possible over the rational numbers or over Z/p, p a prime number.
SINGULAR does not check whether the given polynomial is irreducible! It can be checked in advance with the function `factorize` (see Section 5.1.30 [factorize], page 140).

Example:

```
  //(Q[i]/(i^2+1))[x,y,z]:
  ring Cxyz=(0,i),(x,y,z),dp;
  minpoly=i^2+1;
  i2;  //this is a number, not a poly
↦ -1
```

See Section 5.1.30 [factorize], page 140; Section 4.16 [ring], page 113.

5.3.4 multBound

Type: int

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

Example:

```
multBound = 20;
option();
↦    //options for 'std'-command: multBound
ideal j=std(i);
multBound;
↦ 20
multBound = 0;  //disables multBound
```

See Section 4.3 [int], page 72; Section 5.1.80 [mult], page 172; Section 5.1.89 [option], page 177; Section 5.1.120 [std], page 202.

5.3.5 noether

Type: poly

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

Example:

```
  ring R=32003,(x,y,z),ds;
  ideal i=x2+y12,y13;
  std(i);
↦ _[1]=x2+y12
↦ _[2]=y13
  noether=x11;
  std(i);
↦ _[1]=x2
  noether=0; //disables noether
```

See Section 4.12 [poly], page 107; Section 5.1.120 [std], page 202.

5.3.6 printlevel

Type: int

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

Note: See Section 3.9.1 [Procedures in a library], page 53, for a small example about how this is used for the display of comments while procedures are executed.

Example:

```
  voice;
↦ 1
  printlevel=0;
  dbprint(1);
  printlevel=voice;
  dbprint(1);
↦ 1
```

See Section 5.1.12 [dbprint], page 130; Section 4.3 [int], page 72; Section 5.3.11 [voice], page 224.

5.3.7 short

Type: int

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

Example:

```
  ring r=23,x,dp;
  int save=short;
  short=1;
  2x2,x2;
↦ 2x2 x2
  short=0;
  2x2,x2;
↦ 2*x^2 x^2
  short=save;  //resets short to the previous value
```

See Section 4.3 [int], page 72.

5.3.8 timer

Type: int

Purpose:

1. the CPU time (i.e, user and system time) used for each command is printed if timer >0 , if this time is bigger than a (customizable) minimal time and if `printlevel+1 >= voice` (which is by default true on the SINGULAR top level, but not true while procedures are executed).
2. yields the 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., the default unit of time is one second). The minimal time and timer resolution can be set using the command line options `--min-time` and `--ticks-per-sec` and can be checked using `system("--min-time")` and `system("--ticks-per-sec")`.

How to use `timer` in order to measure the time for a sequence of commands, see example below.

Note for Windows95/98:
The value of the `timer` cannot be used (resp. trusted) when SINGULAR is run under Windows95/98 (this is due to the shortcomings of the Windows95/98 operating system). Use Section 5.3.10 [rtimer], page 224, instead.

Example:

```
  timer=1; // The time of each command is printed
  int t=timer; // initialize t by timer
  ring r=0,(x,y,z),dp;
  poly p=(x+2y+3z+4xy+5xz+6yz)^20;
↦ //used time: 0.74 sec
  // timer as int_expression:
  t=timer-t;
  t;  // yields the time in ticks-per-sec (default 1)
↦ 1
      // since t was initialized by timer
  int tps=system("--ticks-per-sec");
  t/tps; // yields the time in seconds truncated to int
↦ 1
```

See Section 3.1.6 [Command line options], page 18; Section 5.3.6 [printlevel], page 221; Section 5.3.10 [rtimer], page 224; Section 5.1.124 [system], page 205; Section 5.3.11 [voice], page 224.

5.3.9 TRACE

Type: int

Purpose: sets level of debugging.

`TRACE=0` no debugging messages are printed.

`TRACE=1` messages about entering and leaving of procedures are displayed.

`TRACE=3` messages about entering and leaving of procedures together with line numbers are displayed.

`TRACE=4` each line is echoed and the interpretation of commands in this line is suspended until the user presses `RETURN`.

`TRACE` 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 72.

5.3.10 rtimer

Type: int

Purpose: identical to `timer` (see Section 5.3.8 [timer], page 222), except that real times (i.e., wall-clock) times are reported, instead of CPU times. This can be trusted on all operating systems (including Windows95/98).

5.3.11 voice

Type: int

Purpose: shows the nesting level of procedures.

Note: See Section 3.9 [Guidelines for writing a library], page 53, for an small example how this is used for the display of comments while procedures are executed.

Example:

```
  voice;
↦ 1
proc p
{
  voice;
};
p();
↦ 2
```

See Section 5.1.12 [dbprint], page 130; Section 5.1.68 [listvar], page 164; Section 5.3.6 [printlevel], page 221.

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 and the rank of any free module must be smaller than 32768
- the precision of long floating point numbers (for ground field `real`) must be smaller than 32768
- integers (of type `int`) have the limited range from -2147483647 to 2147483647
- floating point numbers (type `number` from field `real`) have a limited range which is machine dependent. A typical range is -1.0e-38 to 1.0e+38. The string representation of overflow and underflow is machine dependent, as well. For example `"Inf"` on Linux, or `"+.+00e+00"` on HPUX.
- the length of an identifier is unlimited but `listvar` displays only the first 20 characters
- statements may not contain more than 10000 tokens
- All input to Singular must be 7-bit clean, i.e. special characters like the the German Umlaute (ä, ö, etc.), or the French accent characters may neither appear as input to SINGULAR, nor in libraries or procedure definitions.

6.2 Major differences to the C programming language

Although many constructs from SINGULAR's programming language are similar to those from the C programming language, there are some subtle differences. Most notably:

6.2.1 No rvalue of increments and assignments

The increment operator `++` (resp. decrement operator `--`) has no rvalue, i.e., cannot be used on the right-hand sides of assignments. So, instead of

```
j = i++;  // WRONG!!!
```

(which results in an error), it must be written

```
i++; j = i;
```

Likewise, an assignment expression does not have a result. Therefore, compound assignments like `i = j = k;` are not allowed and result in an error.

6.2.2 Evaluation of logical expressions

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

Note, that this evaluation is different from the left-to-right, conditional evaluation of logical expressions (as found in most programming languages). For example, in these other languages, the value of `(1 || b)` is determined without ever evaluating `b`. This causes some problems with boolean tests on variables, which might not be defined at evaluation time. For example, the following results in an error, if the variable `i` is undefined:

```
if (defined(i) && i > 0) {} // WRONG!!!
```

This must be written instead as:

```
if (defined(i))
{
  if (i > 0) {}
}
```

However, there are several short work-arounds for this problem:

1. If a variable (say, `i`) is only to be used as a boolean flag, then define (value is TRUE) and undefine (value is FALSE) `i` instead of assigning a value. Using this scheme, it is sufficient to simply write

   ```
   if (defined(i))
   ```

   in order to check whether `i` is TRUE. Use the command `kill` to undefine a variable, i.e. to assign it a FALSE value (see Section 5.1.57 [kill], page 158).
2. If a variable can have more than two values, then define it, if necessary, before it is used for the first time. For example, if the following is used within a procedure

   ```
   if (! defined(DEBUG)) { int DEBUG = 1;}
   ...
   if (DEBUG == 3)  {...}
   if (DEBUG == 2)  {...}
   ...
   ```

   then a user of this procedure does not need to care about the existence of the `DEBUG` variable – this remains hidden from the user. However, if `DEBUG` exists globally, then its local default value is overwritten by its global one.

6.2.3 No case or switch statement

Singular does not offer a `case` (or `switch`) statement. However, it can be imitated in the following way:

```
while (1)
{
   if (choice == choice_1) { ...; break;}
   ...
   if (choice == choice_n) { ...; break;}
   // default case
   ...; break;
}
```

6.2.4 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++) {...}
```

6.2.5 Usage of brackets

In Singular, curly brackets (`{ }`) **must always** be used to enclose the statement body following such constructs like `if`, `else`, `for`, or `while`, even if this block consists of only a single statement. Similarly, in the return statement of a procedure, parentheses (`( )`) **must always** be used to enclose the return value. Even if there is no value to return, parentheses have to be used after a return statement (i.e., `return();`). For example,

```
if (i == 1) return i;    // WRONG!!!!!
```

results in an error. Instead, it must be written as

```
if (i == 1) { return (i); }
```

6.2.6 Behavior of continue

SINGULAR's `continue` construct is only valid inside the body of a `for` or `while` construct. It skips the rest of the loop-body and jumps to the beginning of the block. Unlike the C-construct SINGULAR's `continue` **does not execute the increment statement**. For example,

```
for (int i = 1 ; i<=10; i=i+1)
{
   ...
   if (i==3) { i=8;continue; }
     // skip the rest if i is 3 and
     // continue with the next i: 8
   i;
}
↦ 1
↦ 2
↦ 8
↦ 9
↦ 10
```

6.2.7 Return type of procedures

Although the SINGULAR language is a strongly typed programming language, the type of the return value of a procedure does not need to be specified. As a consequence, the return type of a procedure may vary, i.e., may, for example, depend on the input. However, the return value of such a procedure may then only be assigned to a variable of type `def`.

```
proc type_return (int i)
{
  if (i > 0) {return (i);}
  else {return (list(i));}
}
def t1 = type_return(1);
def t2 = type_return(-1);
typeof(t1); typeof(t2);
↦ int
↦ list
```

Furthermore, it is mandatory to assign the return value of a procedure to a variable of type `def`, if a procedure changes the current ring using the `keepring` command (see Section 5.2.8 [keepring], page 217) 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

   If two numerical constants (i.e., two sequences of digits) are divided using the `/` operator, the surrounding whitespace determines which division to use: if there is no space between the constants and the `/` operator (e.g., "3/2"), both numerical constants are treated as of type `number` and the current ring division is used. If there is at least one space surrounding the `/` operator (e.g., "3 / 2"), both numerical constants are treated as of type `int` and an integer division is performed. To avoid confusion, use the `div` operator instead of `/` for integer division and an explicit type cast to `number` for ring division. Note, that this problem does only occur for divisions of numerical constants.

   ```
     ring r=32002,x,dp;
     3/2;    // ring division
   ↦ -15994
     3 / 2;  // integer division
   ↦ 1
     3 div 2;
   ↦ 1
     number(3) / number(2);
   ↦ -15994
     number a=3;
     number b=2;
     a/b;
   ↦ -15994
     int c=3;
     int d=2;
     c / d;
   ↦ 1
   ```

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 or module. Use `ncols` to determine the actual number of entries in the ideal or module.

5. computations in `qring`

   In order to speed up computations in quotient rings, Singular usually does not reduce polynomials w.r.t. the quotient ideal; rather the given representative is used as long as possible during computations. If it is necessary, reduction is done during standard base computations. To reduce a polynomial `f` by hand w.r.t. the current quotient ideal use the command `reduce(f,std(0))` (see Section 5.1.106 [reduce], page 191).

6. substring selection

   To extract substrings from a `string`, square brackets are used, enclosing either two comma-separated `int`s or an `intvec`. Although two comma-separated `int`s represent an `intvec`, they mean different things in substring access. Square brackets enclosing two `int`s (e.g. `s[2,6]`) return a substring where the first integer denotes the starting position and the second integer denotes the length of the substring. The result is returned as a `string`. Square brackets enclosing an `intvec` (e.g. `s[intvec(2,6)]`) return the characters of the string at the position given by the values of the `intvec`. The result is returned as an expression list of strings.

   ```
     string s = "one-word";
     s[2,6];     // a substring starting at the second char
   ↦ ne-wor
     size(_);
   ↦ 6
     intvec v = 2,6;
     s[v];      // the second and the sixth char
   ↦ n o
     string st = s[v];  // stick together by an assignment
     st;
   ↦ no
     size(_);
   ↦ 2
     v = 2,6,8;
     s[v];
   ↦ n o d
   ```

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 (like `ring`, `std`, ...),
2. a local variable (w.r.t. a procedure),
3. a local ring variable (w.r.t. the current basering locally set in a procedure),
4. a global variable,
5. a global ring variable (w.r.t. the current basering)
6. a monomial consisting of local ring variables written without operators,
7. a monomial consisting of global ring variables written without operators.

Consequently, it is allowed to have general variables with the same name as ring variables. However, the above identifier resolution order must be kept in mind. Otherwise, surprising results may come up.

```
ring r=0,(x,y),dp;
int x;
x*y; // resolved product int*poly, i.e., 0*y
↦ 0
xy; // "xy" is one identifier and resolved to monomial xy
↦ xy
```

For these reasons, we strongly recommend not to use variables which have the same name(s) as ring variables.

Moreover, we strongly recommend not to use ring variables whose name is fully contained in (i.e., is a substring of) another name of a ring variable. Otherwise, effects like the following might occur:

```
ring r=0,(x, x1),dp; // name x is substring of name x1 !!!!!!!!!
x;x1;   // resolved poly x
↦ x
↦ x1
short=0; 2x1; // resolved to monomial 2*x^1 !!!!!!
↦ 2*x
2*x1; // resolved to product 2 times x1
↦ 2*x1
```

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

$$\mathrm{milnor}(f) = \dim_K(K[[x_1,\ldots,x_n]]/\mathrm{jacob}(f)),$$

respectively

$$\mathrm{tjurina}(f) = \dim_K(K[[x_1,\ldots,x_n]]/((f)+\mathrm{jacob}(f)))$$

where `jacob(f)` is the ideal generated by the partials of `f`. `tjurina(f)` is finite, if and only if `f` has an isolated singularity. The same holds for `milnor(f)` if K has characteristic 0. SINGULAR displays -1 if the dimension is infinite.

SINGULAR cannot compute with infinite power series. But it can work in $\mathrm{Loc}_{(x)}K[x_1,\ldots,x_n]$, the localization of $K[x_1,\ldots,x_n]$ at the maximal ideal $(x_1,\ldots,x_n)$. To do this one has to define 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;`). See Section B.2 [Monomial orderings], page 275.

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);
↦ [1023:2]7(2)s8s10s11s12s(3)s13(4)s(5)s14(6)s(7)15--.s(6)-16.-.s(5)17.s(7)\
   s--s18(6).--19-..sH(24)20(3)...21....22....23.--24-
↦ product criterion:10 chain criterion:69
  "The Milnor number of f(11,5,3) for t=0 is", vdim(j);
↦ The Milnor number of f(11,5,3) for t=0 is 250
  j=i+f;    // overwrite j
  j=std(j);
↦ [1023:2]7(3)s8(2)s10s11(3)ss12(4)s(5)s13(6)s(8)s14(9).s(10).15--sH(23)(8)\
   ...16......17.......sH(21)(9)sH(20)16(10).17...........18.......19..----.\
   .sH(19)
↦ product criterion:10 chain criterion:53
  vdim(j);  // compute the Tjurina number for t=0
↦ 195
  t=1;
  f=x^a+y^b+z^(3*c)+x^(c+2)*y^(c-1)+x^(c-1)*y^(c-1)*z3
    +x^(c-2)*y^c*(y^2+t*x)^2;
  ideal i1=jacob(f);
  ideal j1=std(i1);
↦ [1023:2]7(2)s8s10s11s12s13(3)ss(4)s14(5)s(6)s15(7).....s(8)16.s...s(9)..1\
   7............s18(10).....s(11)..-.19.......sH(24)(10).....20...........21\
   ..........22............................23.............................\
   .24.----------.25.26
↦ product criterion:11 chain criterion:83
  "The Milnor number of f(11,5,3) for t=1:",vdim(j1);
↦ The Milnor number of f(11,5,3) for t=1: 248
  vdim(std(j1+f));   // compute the Tjurina number for t=1
↦ [1023:2]7(16)s8(15)s10s11ss(16)-12.s-s13s(17)s(18)s(19)-s(18).-14-s(17)-s\
   (16)ss(17)s15(18)..-s...--.16....-.......s(16).sH(23)s(18)...17..........\
   18.........sH(20)17(17)...................18..........19..---....-.-....\
   .....20.-----...s17(9).........18..............19..-.......20.-......21..\
   .......sH(19)16(5).....18......19.-----
↦ product criterion:15 chain criterion:174
↦ 195
  option(noprot);
```

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:

- load 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 1st, `#[2]` for the 2nd 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 231 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 ground field is different. Compare with `fetch` which works for almost identical rings, e.g., if the rings differ only by the ordering or by the names of the variables and which may be used to rename variables). Integers and strings, however, do not belong to any ring. Once defined they are globally known.
- The result of the computation here (together with the previous one in Section A.1 [Milnor and Tjurina], page 231) shows that (for `t`=0) $\dim_K(\mathrm{Loc}_{(x,y,z)}K[x,y,z]/\mathrm{jacob}(f)) = 250$ (previously computed) while $\dim_K(K[x,y,z]/\mathrm{jacob}(f)) = 536$. Hence `f` has 286 critical points, counted with multiplicity, outside the origin. Moreover, since $\dim_K(\mathrm{Loc}_{(x,y,z)}K[x,y,z]/(\mathrm{jacob}(f)+(f)))$ $= 195 = \dim_K(K[x,y,z]/(\mathrm{jacob}(f)+(f)))$, the affine surface `f`=0 is smooth outside the origin.

```
  ring r1 = 32003,(x,y,z),ds;
  int a,b,c,t=11,5,3,0;
  poly f = x^a+y^b+z^(3*c)+x^(c+2)*y^(c-1)+x^(c-1)*y^(c-1)*z3+
           x^(c-2)*y^c*(y^2+t*x)^2;
  option(noprot);
  timer=1;
  ring r2 = 32003,(x,y,z),dp;
  poly f=imap(r1,f);
  ideal j=jacob(f);
  vdim(std(j));
↦ 536
  vdim(std(j+f));
↦ 195
  timer=0;  // reset timer
```

A.4 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

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

denote the saturation of i with respect to j . This defines, geometrically, the closure of the complement of V(j) in V(i) (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..., 83 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 has been shown, that in complicated examples, more than 95 percent of the time needed for a standard basis computation is used in the computation of the coefficients (in char 0). The representation of long integers with real is demonstrated.

```
timer = 1;                              // activate the timer
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+1\
   1y19
```

```
ideal j = std(i1);
↦ [65535:1]11(2)ss19s20s21s22(3)-23-s27s28s29s30s31s32s33s34s35s36s37s38s39\
   s40s70-
↦ product criterion:1 chain criterion:30
j;
↦ j[1]=264627y39+26244y35-1323135y30-131220y26+1715175y21+164025y17+1830125\
   y16
↦ j[2]=12103947791971846719838321886393392913750065060875xy8-28639152114168\
   3198701331939250003266767738632875y38-31954402206909026926764622877573565\
   78554430672591y37+57436621420822663849721381265738895282846320y36+1657764\
   214948799497573918210031067353932439400y35+213018481589308191195677223898\
   98682697001205500y34+1822194158663066565585991976961565719648069806148y33\
   -4701709279892816135156972313196394005220175y32-1351872269688192267600786\
   97600850686824231975y31-3873063305929810816961516976025038053001141375y30\
   +1325886675843874047990382005421144061861290080000y29+1597720195476063141\
   9467945895542406089526966887310y28-26270181336309092660633348002625330426\
   7126525y27-7586082690893335269027136248944859544727953125y26-867853074106\
   49464602285843351672148965395945625y25-5545808143273594102173252331151835\
   700278863924745y24+19075563013460437364679153779038394895638325y23+548562\
   322715501761058348996776922561074021125y22+157465452677648386073957464715\
   68100780933983125y21-141427912972117622297865423581735950555191156250y20\
   -20711190069445893615213399650035715378169943423125y19+272942733337472665\
   573418092977905322984009750y18+789065115845334505801847294677413365720955\
   3750y17+63554897038491686787729656061044724651089803125y16-22099251729923\
   906699732244761028266074350255961625y14+147937139679655904353579489722585\
   91339027857296625y10
↦ j[3]=5x4+3x2y9+y9
// Compute average coefficient length (=51) by
//   - converting j[2] to a string in order to compute the number
//   of characters
//   - divide this by the number of monomials:
size(string(j[2]))/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;
short=0; // force the long output format
poly p=imap(R0,p);
p;
↦ x*y^8-2.366e-02*y^38-2.640e-01*y^37+4.745e-06*y^36+1.370e-04*y^35+1.760e-\
   03*y^34+1.505e-01*y^33+3.884e-07*y^32-1.117e-05*y^31-3.200e-04*y^30+1.095\
   e-01*y^29+1.320e+00*y^28-2.170e-05*y^27-6.267e-04*y^26-7.170e-03*y^25-4.5\
   82e-01*y^24+1.576e-06*y^23+4.532e-05*y^22+1.301e-03*y^21-1.168e-01*y^20-1\
   .711e+00*y^19+2.255e-05*y^18+6.519e-04*y^17+5.251e-03*y^16-1.826e+00*y^14\
   +1.222e+00*y^10
// Compute a standard basis for the deformed ideal:
setring R0;
j = std(i2);
↦ [65535:1]11(2)ss19s20s21s22(3)-s23(2)s27.28.s29(3)s30.s31ss32sss33sss34ss\
   35--38-
```

```
↦ product criterion:11 chain criterion:21
j;
↦ j[1]=y16
↦ j[2]=65610xy8+17393508y27+7223337y23+545292y19+6442040y18-119790y14+80190\
  y10
↦ j[3]=5x4+3x2y9+y9
vdim(j);
↦ 40
```

A.6 Parameters

Let us deform the above ideal now 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

T^1 , resp. T^2 , of an ideal j usually denote the modules of infinitesimal deformations, resp. of obstructions. In Singular there are procedures `T_1` and `T_2` in `sing.lib` such that `T_1(j)` and `T_2(j)` compute a standard basis of a presentation of these modules. If T_1 and T_2 are finite dimensional K-vector spaces (e.g., for isolated singularities), a basis can be computed by applying `kbase(T_1(j));`, resp. `kbase(T_2(j));`, the dimensions by applying `vdim`. For a complete intersection j the procedure `Tjurina` also computes T_1, but faster (T_2=0 in this case). For a non complete intersection, it is faster to use the procedure `T_12` instead of `T_1` and `T_2`. Type `help T_1;` (or `help T_2;` or `help T_12;`) to obtain more detailed information about these procedures.

We give three examples, the first being a hypersurface, the second a complete intersection, the third no complete intersection:

- load `sing.lib`
- check whether the ideal j is a complete intersection. It is, if number of variables = dimension + minimal number of generators

- compute the Tjurina number
- compute a vector space basis (kbase) of T_1
- compute the Hilbert function of T_1
- create a polynomial encoding the Hilbert series
- compute the dimension of T_2

```
  LIB "sing.lib";
  ring R=32003,(x,y,z),ds;
  // ---------------------------------------
  // hypersurface case (from series T[p,q,r]):
  int p,q,r = 3,3,4;
  poly f = x^p+y^q+z^r+xyz;
  tjurina(f);
↦ 8
  // Tjurina number = 8
  kbase(Tjurina(f));
↦ // Tjurina number = 8
↦ _[1]=z3
↦ _[2]=z2
↦ _[3]=yz
↦ _[4]=xz
↦ _[5]=z
↦ _[6]=y
↦ _[7]=x
↦ _[8]=1
  // ---------------------------------------
  // complete intersection case (from series P[k,l]):
  int k,l =3,2;
  ideal j=xy,x^k+y^l+z2;
  dim(std(j));          // Krull dimension
↦ 1
  size(minbase(j));     // minimal number of generators
↦ 2
  tjurina(j);           // Tjurina number
↦ 6
  module T=Tjurina(j);
↦ // Tjurina number = 6
  kbase(T);             // a sparse output of the k-basis of T_1
↦ _[1]=z*gen(1)
↦ _[2]=gen(1)
↦ _[3]=y*gen(2)
↦ _[4]=x2*gen(2)
↦ _[5]=x*gen(2)
↦ _[6]=gen(2)
  print(kbase(T));      // columns of matrix are a k-basis of T_1
↦ z,1,0,0, 0,0,
↦ 0,0,y,x2,x,1
  // ---------------------------------------
  // general case (cone over rational normal curve of degree 4):
  ring r1=0,(x,y,z,u,v),ds;
  matrix m[2][4]=x,y,z,u,y,z,u,v;
  ideal i=minor(m,2);   // 2x2 minors of matrix m
  module M=T_1(i);       // a presentation matrix of T_1
```

```
↦ // dim T_1 = 4
  vdim(M);              // Tjurina number
↦ 4
  hilb(M);              // display of both Hilbert series
↦ //         4 t^0
↦ //       -20 t^1
↦ //        40 t^2
↦ //       -40 t^3
↦ //        20 t^4
↦ //        -4 t^5
↦
↦ //         4 t^0
↦ // dimension    = 0
↦ // multiplicity = 4
  intvec v1=hilb(M,1);  // first Hilbert series as intvec
  intvec v2=hilb(M,2);  // second Hilbert series as intvec
  v1;
↦ 4,-20,40,-40,20,-4,0
  v2;
↦ 4,0
  v1[3];                // 3rd coefficient of the 1st Hilbert series
↦ 40
  module N=T_2(i);
↦ // dim T_2 = 3
```

```
// In some cases it might be useful to have a polynomial in some ring
// encoding the Hilbert series. This polynomial can then be
// differentiated, evaluated etc. It can be done as follows:
ring H = 0,t,ls;
poly h1;
int ii;
for (ii=1; ii<=size(v1); ii=ii+1)
{
   h1=h1+v1[ii]*t^(ii-1);
}
h1;                   // 1st Hilbert series
↦ 4-20t+40t2-40t3+20t4-4t5
diff(h1,t);           // differentiate  h1
↦ -20+80t-120t2+80t3-20t4
subst(h1,t,1);        // substitute t by 1
↦ 0

// The procedures T_1, T_2, T_12 may be called with two arguments and then
// they return a list with more information (type help T_1; etc.)
// e.g., T_12(i,<any>); returns a list with 9 nonempty objects where
// _[1] = std basis of T_1-module, _[2] = std basis of T_2-module,
// _[3]= vdim of T_1, _[4]= vdim of T_2
setring r1;           // make r1 again the basering
list L = T_12(i,1);
↦ // dim T_1  =  4
↦ // dim T_2  =  3
kbase(L[1]);          // kbase of T_1
↦ _[1]=1*gen(2)
↦ _[2]=1*gen(3)
```

```
↦ _[3]=1*gen(6)
↦ _[4]=1*gen(7)
kbase(L[2]);           // kbase of T_2
↦ _[1]=1*gen(6)
↦ _[2]=1*gen(8)
↦ _[3]=1*gen(9)
L[3];                  // vdim of T_1
↦ 4
L[4];                  // vdim of T_2
↦ 3
```

A.8 Deformations

- The libraries `sing.lib`, resp. `deform.lib`, contain procedures to compute total and base space of 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 $h_1, \ldots, h_r$ represent all 1st order deformations. More precisely, if $I \subset R$ is the ideal generated by $f_1, ..., f_s$, then any infinitesimal deformation of R/I over $K[\varepsilon]/(\varepsilon^2)$ is given by $f + \varepsilon g$, where $f = (f_1, ..., f_s)$, g a K-linear combination of the h_i.
- The procedure `versal` in `deform.lib` computes a formal miniversal deformation up to a certain order which can be prescribed by the user. For a complete intersection the 1st order part is already miniversal.
- The procedure `versal` extends the basering to a new ring with additional deformation parameters which contains the equations for the miniversal base space and the miniversal total space.
- There are default names for the objects created, but the user may also choose his own names.
- If the user sets `printlevel=2;` before running `versal`, some intermediate results are shown. This is useful since `versal` is already complicated and might run for some time on more complicated examples. (type `help versal;`)

We compute for the same examples as in the preceding 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: T_1 and T_2
↦
↦ // 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
↦ // objects belonging to this ring:
↦ // Rs                   [0]  matrix 2 x 1
↦ // Fs                   [0]  matrix 1 x 2
↦ // Js                   [0]  matrix 1 x 0
  listvar(matrix);
↦ // Rs                   [0]  matrix 2 x 1
↦ // Fs                   [0]  matrix 1 x 2
↦ // Js                   [0]  matrix 1 x 0
  // ___ Equations of miniversal base space ___:
  Js;
↦
  // ___ Equations of miniversal total space ___:
  Fs;
↦ Fs[1,1]=xy+Ez+F
↦ Fs[1,2]=y2+z2+x3+Ay+Bx2+Cx+D
  // the miniversal deformation of V(j) is the projection from the
  // miniversal total space to the miniversal base space:
  // { (A,B,C,D,E,F,x,y,z) | xy+F+Ez=0, y2+z2+x3+D+Cx+Bx2+Ay=0 }
  //  --> { (A,B,C,D,E,F) }
  //----------------------------------------------------
  // general case (cone over rational normal curve of degree 4):
  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: T_1 and T_2
↦
↦ // 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 typin\
   g:
↦    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 of last command
↦ // used time: 1 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 zeros on the torus. First over the prime field and then in the finite extension field with p^k elements. In general there will be many more solutions in the second case. (Since the Singular language is interpreted, the evaluation of many `for`-loops is not very fast):

```
  int p=3;  int n=3;  int d=5; int k=2;
  ring rp = p,(x(1..n)),dp;
  int s = size(maxideal(d));
  s;
↦ 21
  // create a dense homogeneous ideal m, all generators of degree d, with
  // generic (random) coefficients:
  ideal m = maxideal(d)*random(p,s,n-2);
  m;
↦ m[1]=x(1)^3*x(2)^2-x(1)*x(2)^4+x(1)^4*x(3)-x(1)^3*x(2)*x(3)+x(1)*x(2)^3*x\
   (3)+x(2)^4*x(3)+x(2)^3*x(3)^2+x(1)*x(2)*x(3)^3+x(1)*x(3)^4-x(3)^5
  // look for zeros on the torus by checking all points (with no component 0)
  // of the affine n-space over the field with p elements :
  ideal mt;
  int i(1..n);                    // initialize integers i(1),...,i(n)
  int l;
  s=0;
  for (i(1)=1;i(1)<p;i(1)=i(1)+1)
  {
    for (i(2)=1;i(2)<p;i(2)=i(2)+1)
    {
      for (i(3)=1;i(3)<p;i(3)=i(3)+1)
      {
        mt=m;
        for (l=1;l<=n;l=l+1)
        {
          mt=subst(mt,x(l),i(l));
        }
        if (size(mt)==0)
        {
          "solution:",i(1..n);
```

```
          s=s+1;
        }
      }
    }
  }
↦ solution: 1 1 2
↦ solution: 1 2 1
↦ solution: 1 2 2
↦ solution: 2 1 1
↦ solution: 2 1 2
↦ solution: 2 2 1
  "//",s,"solutions over GF("+string(p)+")";
↦ // 6 solutions over GF(3)
  // Now go to the field with p^3 elements:
  // As long as there is no map from Z/p to the field with p^3 elements
  // implemented, use the following trick: convert the ideal to be mapped
  // to the new ring to a string and then execute this string in the
  // new ring
  string ms="ideal m="+string(m)+";";
  ms;
↦ ideal m=x(1)^3*x(2)^2-x(1)*x(2)^4+x(1)^4*x(3)-x(1)^3*x(2)*x(3)+x(1)*x(2)^\
   3*x(3)+x(2)^4*x(3)+x(2)^3*x(3)^2+x(1)*x(2)*x(3)^3+x(1)*x(3)^4-x(3)^5;
  // define a ring rpk with p^k elements, call the primitive element z. Hence
  // 'solution exponent: 0 1 5' means that (z^0,z^1,z^5) is a solution
  ring rpk=(p^k,z),(x(1..n)),dp;
  rpk;
↦ //   # 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
  execute(ms);
  s=0;
  ideal mt;
  for (i(1)=0;i(1)<p^k-1;i(1)=i(1)+1)
  {
    for (i(2)=0;i(2)<p^k-1;i(2)=i(2)+1)
    {
      for (i(3)=0;i(3)<p^k-1;i(3)=i(3)+1)
      {
        mt=m;
        for (l=1;l<=n;l=l+1)
        {
          mt=subst(mt,x(l),z^i(l));
        }
        if (size(mt)==0)
        {
          "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: 0 4 7
↦ solution exponent: 1 1 3
↦ solution exponent: 1 1 5
↦ solution exponent: 1 1 7
↦ 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: 2 6 7
↦ solution exponent: 3 3 1
↦ solution exponent: 3 3 5
↦ solution exponent: 3 3 7
↦ solution exponent: 3 4 3
↦ solution exponent: 3 6 3
↦ solution exponent: 3 7 0
↦ solution exponent: 3 7 2
↦ solution exponent: 3 7 3
↦ solution exponent: 3 7 7
↦ 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: 4 7 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 5 7
↦ 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
↦ solution exponent: 6 7 6
↦ solution exponent: 7 0 7
↦ solution exponent: 7 2 7
↦ solution exponent: 7 3 3
↦ solution exponent: 7 3 4
↦ solution exponent: 7 3 6
↦ solution exponent: 7 3 7
↦ solution exponent: 7 7 1
↦ solution exponent: 7 7 3
↦ solution exponent: 7 7 5
  "//",s,"solutions over GF("+string(p^k)+")";
↦ // 72 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 precedes 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 needs not be a standard basis in the given ordering. Hence, there may be cases where the first method is the preferred one.

WARNING: In the case of a local or a mixed ordering, elimination needs special care. f may be considered as a map of germs $f : (k^r, 0) \to (k^n, 0)$, but even if this map germ is finite, we are in general not able to compute the image germ because for this we would need an implementation of the Weierstrass preparation theorem. What we can compute, and what `eliminate` actually does, is the following: let V(J) be the zero-set of J in $k^r \times (k^n, 0)$, then the closure of the image of V(J) under the projection

$$\mathrm{pr} : k^r \times (k^n, 0) \to (k^n, 0)$$

can be computed. Note that this germ contains also those components of V(J) which meet the fiber of pr outside the origin. This is achieved by an ordering with the block of t-variables having a global ordering (and preceding the x-variables) and the x-variables having a local ordering. In a local situation we propose `eliminate` with ordering ls.

In any case, if the input is weighted homogeneous (=quasihomogeneous), the weights given to the variables should be chosen accordingly. SINGULAR offers a function `weight` which proposes, given an ideal or module, integer weights for the variables, such that the ideal, resp. module, is as homogeneous as possible with respect to these weights. The function finds correct weights, if the input is weighted homogeneous (but is rather slow for many variables). In order to check, whether the input is quasihomogeneous, use the function `qhweight`, which returns an intvec of correct weights if the input is quasihomogeneous and an intvec of zeros otherwise.

Let us give two examples:

1. First we compute the equations of the simple space curve T[7]′ consisting of two tangential cusps given in parametric form.
2. We compute weights for the equations such that the equations are quasihomogeneous w.r.t. these weights.
3. Then we compute the tangent developable of the rational normal curve in P^4.

```
  // 1. Compute equations of curve given in parametric form:
  // Two transversal cusps in (k^3,0):
  ring r1 = 0,(t,x,y,z),ls;
  ideal i1 = x-t2,y-t3,z;        // parametrization of the first branch
  ideal i2 = y-t2,z-t3,x;        // parametrization of the second branch
  ideal j1 = eliminate(i1,t);
  j1;                            // equations of the first branch
↦ j1[1]=z
↦ j1[2]=y2-x3
  ideal j2 = eliminate(i2,t);
  j2;                            // equations of the second branch
↦ j2[1]=z2-y3
↦ j2[2]=x
  // Now map to a ring with only x,y,z as variables and compute the
  // intersection of j1 and j2 there:
  ring r2 = 0,(x,y,z),ds;
  ideal j1= imap(r1,j1);         // imap is a convenient ringmap for
  ideal j2= imap(r1,j2);         // inclusions and projections of rings
  ideal i = intersect(j1,j2);
  i;                             // equations of both branches
↦ i[1]=z2-y3+x3y
↦ i[2]=xz
↦ i[3]=xy2-x4
↦ i[4]=x3z
  //
  // 2. Compute the weights:
  intvec v= qhweight(i);         // compute weights
  v;
↦ 4,6,9
  //
  // 3. Compute the tangent developable
  // The tangent developable of a projective variety given parametrically
  // by F=(f1,...,fn) : P^r --> P^n is the union of all tangent spaces
  // of the image. The tangent space at a smooth point F(t1,...,tr)
  // is given as the image of the tangent space at (t1,...,tr) under
  // the tangent map (affine coordinates)
  //   T(t1,...,tr): (y1,...,yr) --> jacob(f)*transpose((y1,...,yr))
  // where jacob(f) denotes the jacobian matrix of f with respect to the
  // t's evaluated at the point (t1,...,tr).
  // Hence we have to create the graph of this map and then to eliminate
```

```
  // the t's and y's.
  // The rational normal curve in P^4 is given as the image of
  //        F(s,t) = (s4,s3t,s2t2,st3,t4)
  // each component being homogeneous of degree 4.
  ring P = 0,(s,t,x,y,a,b,c,d,e),dp;
  ideal M = maxideal(1);
  ideal F = M[1..2];     // take the 1st two generators of M
  F=F^4;
  // simplify(...,2); deletes 0-columns
  matrix jac = simplify(jacob(F),2);
  ideal T = x,y;
  ideal J = jac*transpose(T);
  ideal H = M[5..9];
  ideal i = H-J;         // this is tricky: difference between two
                         // ideals is not defined, but between two
                         // matrices. By automatic type conversion
                         // the ideals are converted to matrices,
                         // subtracted and afterwards converted
                         // to an ideal. Note that '+' is defined
                         // and adds (concatenates) two ideals
  i;
↦ i[1]=-4s3x+a
↦ i[2]=-3s2tx-s3y+b
↦ i[3]=-2st2x-2s2ty+c
↦ i[4]=-t3x-3st2y+d
↦ i[5]=-4t3y+e
  // Now we define a ring with product ordering and weights 4
  // for the variables a,...,e.
  // Then we map i from P to P1 and eliminate s,t,x,y from i.
  ring P1 = 0,(s,t,x,y,a,b,c,d,e),(dp(4),wp(4,4,4,4,4));
  ideal i = fetch(P,i);
  ideal j= eliminate(i,stxy);    // equations of tangent developable
  j;
↦ j[1]=3c2-4bd+ae
↦ j[2]=2bcd-3ad2-3b2e+4ace
↦ j[3]=8b2d2-9acd2-9b2ce+12ac2e-2abde
  // We can use the product ordering to eliminate s,t,x,y from i
  // by a std-basis computation.
  // We need proc 'nselect' from elim.lib.
  LIB "elim.lib";
  j = std(i);                    // compute a std basis j
  j = nselect(j,1,4);            // select generators from j not
  j;                             // containing variable 1,...,4
↦ j[1]=3c2-4bd+ae
↦ j[2]=2bcd-3ad2-3b2e+4ace
↦ j[3]=8b2d2-9acd2-9b2ce+12ac2e-2abde
```

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 5.1.109 [res], page 193
computes a free resolution of an ideal or module using a heuristically chosen method. This is the preferred method to compute free resolutions of ideals or modules.

`lres` Section 5.1.69 [lres], page 165
computes a free resolution of an ideal or module with La Scala's method. The input needs to be homogeneous.

`mres` Section 5.1.78 [mres], page 171
computes a minimal free resolution of an ideal or module with the syzygy method.

`sres` Section 5.1.118 [sres], page 200
computes a free resolution of an ideal or module with Schreyer's method. The input has to be a standard basis.

`nres` Section 5.1.85 [nres], page 175
computes a free resolution of an ideal or module with the standard basis method.

`minres` Section 5.1.74 [minres], page 168
minimizes a free resolution of an ideal or module.

`syz` Section 5.1.125 [syz], page 207
computes the first syzygy module.

`res(i,r)`, `lres(i,r)`, `sres(i,r)`, `mres(i,r)`, `nres(i,r)` compute the first r modules of the resolution of i, resp. the full resolution if r=0 and the basering is not a qring. See the manual for a precise description of these commands.
Note: The command `betti` does not require a minimal resolution for the minimal betti numbers.

Now let's look at an example which uses resolutions: The Hilbert-Burch theorem says that the ideal i of a reduced curve in K^3 has a free resolution of length 2 and that i is given by the 2x2 minors of the 2nd matrix in the resolution. We test this for two transversal cusps in K^3. Afterwards we compute the resolution of the ideal j of the tangent developable of the rational normal curve in P^4 from above. Finally we demonstrate the use of the type `resolution` in connection with the `lres` command.

```
  // Two transversal cusps in (k^3,0):
  ring r2 =0,(x,y,z),ds;
  ideal i =z2-1y3+x3y,xz,-1xy2+x4,x3z;
  resolution rs=mres(i,0);   // computes a minimal resolution
  rs;                        // the standard representation of complexes
↦   1       3       2
↦ r2 <--  r2 <--  r2
↦
↦ 0       1       2
↦
    list resi=rs;            // convertion to a list
  print(resi[1]);            // the 1st module is i minimized
↦ xz,
↦ z2-y3+x3y,
↦ xy2-x4
  print(resi[2]);            // the 1st syzygy module of i
↦ -z,-y2+x3,
```

```
↦ x, 0,
↦ y, z
  resi[3];                   // the 2nd syzygy module of i
↦ _[1]=0
  ideal j=minor(resi[2],2);
  reduce(j,std(i));          // check whether j is contained in i
↦ _[1]=0
↦ _[2]=0
↦ _[3]=0
  size(reduce(i,std(j)));    // check whether i is contained in j
↦ 0
  // size(<ideal>) counts the non-zero generators
  // --------------------------------------------
  // The tangent developable of the rational normal curve in P^4:
  ring P = 0,(a,b,c,d,e),dp;
  ideal j= 3c2-4bd+ae, -2bcd+3ad2+3b2e-4ace,
           8b2d2-9acd2-9b2ce+9ac2e+2abde-1a2e2;
  resolution rs=mres(j,0);
  rs;
↦  1      2      1
↦ P <--  P <--  P
↦
↦ 0      1      2
↦
  list L=rs;
  print(L[2]);
↦ 2bcd-3ad2-3b2e+4ace,
↦ -3c2+4bd-ae
  // create an intmat with graded betti numbers
  intmat B=betti(rs);
  // this gives a nice output of betti numbers
  print(B,"betti");
↦            0     1     2
↦ ------------------------
↦     0:     1     -     -
↦     1:     -     1     -
↦     2:     -     1     -
↦     3:     -     -     1
↦ ------------------------
↦ total:     1     2     1
  // the user has access to all betti numbers
  // the 2-nd column of B:
  B[1..4,2];
↦ 0 1 1 0
  ring cyc5=32003,(a,b,c,d,e,h),dp;
  ideal i=
  a+b+c+d+e,
  ab+bc+cd+de+ea,
  abc+bcd+cde+dea+eab,
  abcd+bcde+cdea+deab+eabc,
  h5-abcde;
  resolution rs=lres(i,0);   //computes the resolution according La Scala
  rs;                        //the shape of the minimal resolution
```

```
↦     1         5         10         10         5         1
↦ cyc5 <--  cyc5 <--  cyc5 <--   cyc5 <--   cyc5 <--  cyc5
↦
↦ 0         1         2          3          4         5
↦ resolution not minimized yet
↦
  print(betti(rs),"betti");  //shows the Betti-numbers of cyclic 5
↦            0     1     2     3     4     5
↦ ------------------------------------------
↦     0:     1     1     -     -     -     -
↦     1:     -     1     1     -     -     -
↦     2:     -     1     1     -     -     -
↦     3:     -     1     2     1     -     -
↦     4:     -     1     2     1     -     -
↦     5:     -     -     2     2     -     -
↦     6:     -     -     1     2     1     -
↦     7:     -     -     1     2     1     -
↦     8:     -     -     -     1     1     -
↦     9:     -     -     -     1     1     -
↦    10:     -     -     -     -     1     1
↦ ------------------------------------------
↦ total:     1     5    10    10     5     1
  dim(rs);                   //the homological dimension
↦ 4
  size(list(rs));            //gets the full (non-reduced) resolution
↦ 6
  minres(rs);                //minimizes the resolution
↦     1         5         10         10         5         1
↦ cyc5 <--  cyc5 <--  cyc5 <--   cyc5 <--   cyc5 <--  cyc5
↦
↦ 0         1         2          3          4         5
↦
  size(list(rs));            //gets the minimized resolution
↦ 6
```

A.12 Computation of Ext

We start by showing how to calculate the n -th Ext group of an ideal. The ingredients to do this are by the definition of Ext the following: calculate a (minimal) resolution at least up to length n , apply the Hom-functor, and calculate the n -th homology group, that is form the quotient ker/Im in the resolution sequence.

The Hom functor is given simply by transposing (hence dualizing) the module or the corresponding matrix with the command `transpose`. The image of the $(n-1)$ -st map is generated by the columns of the corresponding matrix. To calculate the kernel apply the command `syz` at the $(n-1)$ -st transposed entry of the resolution. Finally, the quotient is obtained by the command `modulo`, which gives for two modules A = ker, B = Im the module of relations of $A/(A\cap B)$ in the usual way. As we have a chain complex this is obviously the same as ker/Im.

We collect these statements in the following short procedure:

```
proc ext(int n, ideal I)
{
  resolution rs = mres(I,n+1);
```

```
  module tAn    = transpose(rs[n+1]);
  module tAn_1  = transpose(rs[n]);
  module ext_n  = modulo(syz(tAn),tAn_1);
  return(ext_n);
}
```

Now consider the following example:

```
ring r5 = 32003,(a,b,c,d,e),dp;
ideal I = a2b2+ab2c+b2cd, a2c2+ac2d+c2de,a2d2+ad2e+bd2e,a2e2+abe2+bce2;
print(ext(2,I));
↦ 1,0,0,0,0,0,0,
↦ 0,1,0,0,0,0,0,
↦ 0,0,1,0,0,0,0,
↦ 0,0,0,1,0,0,0,
↦ 0,0,0,0,1,0,0,
↦ 0,0,0,0,0,1,0,
↦ 0,0,0,0,0,0,1
ext(3,I);   // too big to be displayed here
```

The library `homolog.lib` contains several procedures for computing Ext-modules and related modules, which are much more general and sophisticated 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, $R=Loc_m K[x,y]/(x^2-y^3)$, $m=(x,y)$. To compute $\mathrm{Ext}^1(m,m)$ we have to apply `Ext(1,syz(m),syz(m))` with `syz(m)` the first syzygy module of m , which is isomorphic to $\mathrm{Ext}^2(K,K)$.
- $\mathrm{Ext}^k(R/i,R)$ for some ideal i and with an extra option.

```
  LIB "homolog.lib";
  ring R=0,(x,y),ds;
  ideal i=x2-y3;
  qring q = std(i);      // defines the quotient ring Loc_m k[x,y]/(x2-y3)
  ideal m = maxideal(1);
  module T1K = Ext(1,m,m);  // computes Ext^1(R/m,R/m)
↦ // degree of Ext^1:
↦ 0
↦ 
  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 Ext^2(R/m,R/m)
↦ // Computing Ext^2 (help Ext; gives an explanation):
↦ // Let 0<--coker(M)<--F0<--F1<--F2<--... be a resolution of coker(M),
↦ // and 0<--coker(N)<--G0<--G1 a presentation of coker(N),
```

```
↦ // then Hom(F2,G0)-->Hom(F3,G0) is given by:
↦ y2,x,
↦ x, y
↦ // and Hom(F1,G0) + Hom(F2,G1)-->Hom(F2,G0) is given by:
↦ -y,x,  x,0,y,0,
↦ x, -y2,0,x,0,y
↦
↦ // degree of Ext^2:
↦ 0
↦
  print(std(T2K));
↦ -y2,0,x,0,y,
↦ 0,  x,0,y,0,
↦ 1,  0,0,0,0
  printlevel=0;
  module E = Ext(1,syz(m),syz(m));
↦ // degree of Ext^1:
↦ 0
↦
  print(std(E));
↦ -y,x, 0, 0,0,x,0,y,
↦ 0, -y,-y,0,x,0,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
  //The matrices which we have just computed are presentation matrices
  //of the modules T2K and E. Hence we may ignore those columns
  //containing 1 as an entry and see that T2K and E are isomorphic
  //as expected, but differently presented.
  //-------------------------------------------
  ring S=0,(x,y,z),dp;
  ideal  i = x2y,y2z,z3x;
  module E = Ext_R(2,i);
↦ // degree of Ext^2:
↦ 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:
↦ 0
↦
  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.

Now let us compute this for a family of curves. We need the library `elim.lib` for saturation and `sing.lib` for the singular locus.

```
  LIB "elim.lib";
  LIB "sing.lib";
  // Affine polar curve:
  ring R = 0,(x,z,t),dp;              // global ordering dp
  poly f = z5+xz3+x2-tz6;
  dim_slocus(f);                      // dimension of singular locus
↦ 1
  ideal j = diff(f,x),diff(f,z);
  dim(std(j));                        // dim V(j)
↦ 1
  dim(std(j+ideal(f)));               // V(j,f) also 1-dimensional
↦ 1
  // j defines a curve, but to get the polar curve we must remove the
  // branches contained in f=0 (they exist since dim V(j,f) = 1). This
  // gives the polar curve set theoretically. But for the structure we
  // may take either j:f or j:f^k for k sufficiently large. The first is
  // just the ideal quotient, the second the iterated ideal quotient
  // or saturation. In our case both coincide.
  ideal q = quotient(j,ideal(f));     // ideal quotient
  ideal qsat = sat(j,f)[1];           // saturation, proc from elim.lib
  ideal sq = std(q);
  dim(sq);
↦ 1
  // 1-dimensional, hence q defines the affine polar curve
  //
  // to check that q and qsat are the same, we show both inclusions, i.e.,
  // both reductions must give the 0-ideal
  size(reduce(qsat,sq));
↦ 0
  size(reduce(q,std(qsat)));
```

```
↦ 0
  qsat;
↦ qsat[1]=12zt+3z-10
↦ qsat[2]=5z2+12xt+3x
↦ qsat[3]=144xt2+72xt+9x+50z
  // We see that the affine polar curve does not pass through the origin,
  // hence we expect the local polar "curve" to be empty
  // ------------------------------------------------
  // Local polar curve:
  ring r = 0,(x,z,t),ds;              // local ordering ds
  poly f = z5+xz3+x2-tz6;
  ideal j = diff(f,x),diff(f,z);
  dim(std(j));                        // V(j) 1-dimensional
↦ 1
  dim(std(j+ideal(f)));               // V(j,f) also 1-dimensional
↦ 1
  ideal q = quotient(j,ideal(f));     // ideal quotient
  q;
↦ q[1]=1
  // The local polar "curve" is empty, i.e., V(j) is contained in V(f)
  // ------------------------------------------------
  // Projective polar curve: (we need "sing.lib" and "elim.lib")
  ring P = 0,(x,z,t,y),dp;            // global ordering dp
  poly f = z5y+xz3y2+x2y4-tz6;
                                      // but consider t as parameter
  dim_slocus(f);              // projective 1-dimensional singular locus
↦ 2
  ideal j = diff(f,x),diff(f,z);
  dim(std(j));                        // V(j), projective 1-dimensional
↦ 2
  dim(std(j+ideal(f)));               // V(j,f) also projective 1-dimensional
↦ 2
  ideal q = quotient(j,ideal(f));
  ideal qsat = sat(j,f)[1];           // saturation, proc from elim.lib
  dim(std(qsat));
↦ 2
  // projective 1-dimensional, hence q and/or qsat define the projective
  // polar curve. In this case, q and qsat are not the same, we needed
  // 2 quotients.
  // Let us check both reductions:
  size(reduce(qsat,std(q)));
↦ 4
  size(reduce(q,std(qsat)));
↦ 0
  // Hence q is contained in qsat but not conversely
  q;
↦ q[1]=12zty+3zy-10y2
↦ q[2]=60z2t-36xty-9xy-50zy
  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 a generic symmetric $(n \times n)$-matrix. We do this by computing the resolution over the polynomial ring. Then, by the Auslander-Buchsbaum formula, the depth is equal to the number of variables minus the length of a minimal resolution. This example was suggested by U. Vetter in order to check whether his bound $\mathrm{depth}(\mathrm{D}_k(R)) \geq m(m+1)/2+m-1$ could be improved.

```
LIB "matrix.lib"; LIB "sing.lib";
int n = 4;
int m = 3;
int N = n*(n+1)/2;           // will become number of variables
ring R = 32003,x(1..N),dp;
matrix X = symmat(n);        // proc from matrix.lib
                             // creates the symmetric generic nxn matrix
print(X);
↦ x(1),x(2),x(3),x(4),
↦ x(2),x(5),x(6),x(7),
↦ x(3),x(6),x(8),x(9),
↦ x(4),x(7),x(9),x(10)
ideal J = minor(X,m);
J=std(J);
// Kaehler differentials D_k(R)
// of R=k[x1..xn]/J:
module D = J*freemodule(N)+transpose(jacob(J));
ncols(D);
↦ 110
nrows(D);
↦ 10
//
// Note: D is a submodule with 110 generators of a free module
// of rank 10 over a polynomial ring in 10 variables.
// Compute a full resolution of D with sres.
// This takes about 17 sec on a Mac PB 520c and 2 sec an a HP 735
int time = timer;
module sD = std(D);
list Dres = sres(sD,0);                 // the full resolution
timer-time;                             // time used for std + sres
↦ 1
intmat B = betti(Dres);
```

```
  print(B,"betti");
↦           0     1     2     3     4     5     6
↦ ------------------------------------------------
↦     0:    10     -     -     -     -     -     -
↦     1:     -    10     -     -     -     -     -
↦     2:     -    84   144    60     -     -     -
↦     3:     -     -    35    80    60    16     1
↦ ------------------------------------------------
↦ total:    10    94   179   140    60    16     1
  N-ncols(B)+1;                          // the desired depth
↦ 4
```

A.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 create automatically the ideal of cyclic roots over the basering with n variables. The procedure assumes that the variables consist of a single letter each (hence no indexed variables are allowed; the procedure `cyclic` in `poly.lib` does not have this restriction). Then we compute a standard basis of this ideal and some numerical information. (This ideal is used as a classical benchmark for standard basis computations).

```
// We call the procedure 'cyclic':
proc cyclic (int n)
{
   string vs = varstr(basering)+varstr(basering);
   int c=find(vs,",");
   while ( c!=0 )
   {
      vs=vs[1,c-1]+vs[c+1,size(vs)];
      c=find(vs,",");
   }
   string t,s;
   int i,j;
   for ( j=1; j<=n-1; j=j+1 )
   {
      t="";
      for ( i=1; i <=n; i=i+1 )
      {
         t = t + vs[i,j] + "+";
      }
      t = t[1,size(t)-1] + ","+newline;
      s=s+t;
   }
   s=s+vs[1,n]+"-1";
   return (s);
}

ring r=0,(a,b,c,d,e),lp;         // basering, char 0, lex ordering
string sc=cyclic(nvars(basering));
sc;                              // the string of the ideal
↦ a+b+c+d+e,
↦ ab+bc+cd+de+ea,
↦ abc+bcd+cde+dea+eab,
↦ abcd+bcde+cdea+deab+eabc,
↦ abcde-1
execute("ideal i="+sc+";");      // this defines the ideal of cyclic roots
i;
↦ i[1]=a+b+c+d+e
↦ i[2]=ab+bc+cd+ae+de
↦ i[3]=abc+bcd+abe+ade+cde
↦ i[4]=abcd+abce+abde+acde+bcde
↦ i[5]=abcde-1
timer=1;
ideal j=std(i);
↦ //used time: 7.5 sec
size(j);                         // number of elements in the std basis
↦ 11
degree(j);
↦ // codimension = 5
↦ // dimension   = 0
↦ // degree      = 70
```

A.17 G_a -Invariants

We work in characteristic 0 and use the Lie algebra generated by one vector field of the form $\sum x_i \partial/\partial x_{i+1}$.

```
LIB "ainvar.lib";
int n=5;
int i;
ring s=32003,(x(1..n)),wp(1,2,3,4,5);
// definition of the vector field m=sum m[i,1]*d/dx(i)
matrix m[n][1];
for (i=1;i<=n-1;i=i+1)
{
   m[i+1,1]=x(i);
}
// computation of the ring of invariants
ideal in=invariantRing(m,x(2),x(1),0);
in;   //invariant ring is generated by 5 invariants
↦ in[1]=x(1)
↦ in[2]=x(2)^2-2*x(1)*x(3)
↦ in[3]=x(3)^2-2*x(2)*x(4)+2*x(1)*x(5)
↦ in[4]=x(2)^3-3*x(1)*x(2)*x(3)+3*x(1)^2*x(4)
↦ in[5]=x(3)^3-3*x(2)*x(3)*x(4)-15997*x(1)*x(4)^2+3*x(2)^2*x(5)-6*x(1)*x(3)\
   *x(5)
ring q=32003,(x,y,z,u,v,w),dp;
matrix m[6][1];
m[2,1]=x;
m[3,1]=y;
m[5,1]=u;
m[6,1]=v;
// the vector field is: xd/dy+yd/dz+ud/dv+vd/dw
ideal in=invariantRing(m,y,x,0);
in; //invariant ring is generated by 6 invariants
↦ in[1]=x
↦ in[2]=u
↦ in[3]=v2-2uw
↦ in[4]=zu-yv+xw
↦ in[5]=yu-xv
↦ in[6]=y2-2xz
```

A.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
  print(A*A*A*A); // the fourth power of A is 1
↦ 1,0,0,
↦ 0,1,0,
↦ 0,0,1
  // Use the first method to compute the invariants of G:
  matrix B(1..3);
  B(1..3)=invariant_ring(A);
  // SINGULAR returns 2 matrices, the first containing
  // primary invariants and the second secondary
  // invariants, i.e., module generators over a Noetherian
  // normalization
  // the third result are the irreducible secondary invariants
  // if the Molien series was available
  print(B(1));
↦ z2,x2+y2,x2y2
  print(B(2));
↦ 1,xyz,x2z-y2z,x3y-xy3
  print(B(3));
↦ xyz,x2z-y2z,x3y-xy3
  // Use the second method,
  // with random numbers between -1 and 1:
  B(1..3)=invariant_ring_random(A,1);
  print(B(1..3));
↦ z2,x2+y2,x4+y4-z4
↦ 1,xyz,x2z-y2z,x3y-xy3
↦ xyz,x2z-y2z,x3y-xy3
```

A.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 s =2,(x,y),dp;
  poly g = (y4+x8)*(x2+y2);
  factorize(g);
↦ [1]:
↦    _[1]=1
↦    _[2]=x+y
↦    _[3]=x2+y
↦ [2]:
↦    1,2,4
```

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 most probably be the same in characteristic 0.

We compute the Hamburger-Noether expansion of a plane curve singularity 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 the

- characteristic exponents,
- Puiseux pairs (of a complex model),
- degree of the conductor,
- delta invariant,
- generators of the 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
↦  generators of semigroup   : 4,6,13
↦  Puiseux pairs             : (3,2)(7,2)
↦  degree of the conductor   : 16
↦  delta invariant           : 8
↦  sequence of multiplicities: 4,2,2,1
  // invariants(hn);  returns the invariants as list
  // partial parametrization of f: param takes the first variable
  // as infinite except the ring has more than 2 variables. Then
  // the 3rd variable is chosen.
  param(hn);
↦ // ** Warning: result is exact up to order 5 in x and 7 in y !
↦ _[1]=1/16x4-3/16x5+1/4x7
↦ _[2]=1/64x6-5/64x7+3/32x8+1/16x9-1/8x10
  ring extring=0,(x,y,t),ds;
  poly f=x3+2xy2+y2;
  list hn=develop(f,-1);
↦  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 of 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,
↦ //         basering has changed to HNEring
  show(hn[1][1]);     // Hamburger-Noether matrix of 1st branch
↦ // matrix, 3x3
↦ 0,x,0,
↦ 0,1,x,
↦ 0,1,-1
  displayInvariants(hn);
↦  --- invariants of branch number 1 : ---
↦  characteristic exponents  : 4,6,7
↦  generators of semigroup   : 4,6,13
↦  Puiseux pairs            : (3,2)(7,2)
↦  degree of the conductor   : 16
↦  delta invariant          : 8
↦  sequence of multiplicities: 4,2,2,1
↦
↦  --- invariants of branch number 2 : ---
↦  characteristic exponents  : 2,5
↦  generators of semigroup   : 2,5
↦  Puiseux pairs            : (5,2)
↦  degree of the conductor   : 4
↦  delta invariant          : 2
↦  sequence of multiplicities: 2,2,1
↦
↦  -------------- intersection multiplicities : --------------
↦
↦ branch |    2
↦ -------+-----
↦     1  |   12
  param(hn[2]);      // parametrization of 2nd branch
↦ _[1]=x2
↦ _[2]=x5
```

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` and `primdecSY` is returned as a list of pairs of ideals, where the second ideal form the prime ideal and the first ideal form the corresponding primary ideal.

```
  LIB "primdec.lib";
  ring r = 0,(a,b,c,d,e,f),dp;
  ideal i= f3, ef2, e2f, bcf-adf, de+cf, be+af, e3;
  primdecGTZ(i);
↦ [1]:
↦    [1]:
↦       _[1]=f
↦       _[2]=e
↦    [2]:
↦       _[1]=f
↦       _[2]=e
↦ [2]:
↦    [1]:
↦       _[1]=f3
↦       _[2]=ef2
↦       _[3]=e2f
↦       _[4]=e3
↦       _[5]=de+cf
↦       _[6]=be+af
↦       _[7]=-bc+ad
↦    [2]:
↦       _[1]=f
↦       _[2]=e
↦       _[3]=-bc+ad
  // We consider now the ideal J of the base space of the
  // miniversal deformation of the cone over the rational
  // normal curve computed in section *8* and compute
  // its primary decomposition.
  ring R = 0,(A,B,C,D),dp;
  ideal J = CD, BD+D2, AD;
  primdecGTZ(J);
↦ [1]:
↦    [1]:
↦       _[1]=D
↦    [2]:
↦       _[1]=D
↦ [2]:
↦    [1]:
↦       _[1]=C
↦       _[2]=B+D
↦       _[3]=A
↦    [2]:
↦       _[1]=C
↦       _[2]=B+D
↦       _[3]=A
  // We see that there are two components which are both
  // prime, even linear subspaces, one 3-dimensional,
  // the other 1-dimensional.
  // (This is Pinkhams example and was the first known
  // surface singularity with two components of
  // different dimensions)
```

```
//
// Let us now produce an embedded component in the last
// example, compute the minimal associated primes and
// the radical. We use the Characteristic set methods
// from prim_dec.lib.
J = intersect(J,maxideal(3));
// The following shows that the maximal ideal defines an embedded
// (prime) component.
primdecSY(J);
↦ [1]:
↦    [1]:
↦       _[1]=D
↦    [2]:
↦       _[1]=D
↦ [2]:
↦    [1]:
↦       _[1]=C
↦       _[2]=B+D
↦       _[3]=A
↦    [2]:
↦       _[1]=C
↦       _[2]=B+D
↦       _[3]=A
↦ [3]:
↦    [1]:
↦       _[1]=D2
↦       _[2]=C2
↦       _[3]=B2
↦       _[4]=AB
↦       _[5]=A2
↦       _[6]=BCD
↦       _[7]=ACD
↦    [2]:
↦       _[1]=D
↦       _[2]=C
↦       _[3]=B
↦       _[4]=A
minAssChar(J);
↦ [1]:
↦    _[1]=C
↦    _[2]=B+D
↦    _[3]=A
↦ [2]:
↦    _[1]=D
radical(J);
↦ _[1]=CD
↦ _[2]=BD+D2
↦ _[3]=AD
```

A.22 Normalization

The normalization will be computed for a reduced ring R/I . The result is a list of rings; ideals are always called `norid` in the rings of this list. The normalization of R/I is the product of the factor rings of the rings in the list divided out by the ideals `norid`.

```
  LIB "normal.lib";
  // ----- first example: rational quadruple point -----
  ring R=32003,(x,y,z),wp(3,5,15);
  ideal I=z*(y3-x5)+x10;
  list pr=normal(I);
↦
↦ // 'normal' created a list of 1 ring(s).
↦ // nor[1+1] is the delta-invariant in case of choose=wd.
↦ // To see the rings, type (if the name of your list is nor):
↦      show( nor);
↦ // To access the 1-st ring and map (similar for the others), type:
↦      def R = nor[1]; setring R;  norid; normap;
↦ // R/norid is the 1-st ring of the normalization and
↦ // normap the map from the original basering to R/norid
  def S=pr[1];
  setring S;
  norid;
↦ norid[1]=T(2)*T(3)-T(1)*T(4)
↦ norid[2]=T(1)^7-T(1)^2*T(3)+T(2)*T(5)
↦ norid[3]=T(1)^2*T(5)-T(2)*T(4)
↦ norid[4]=T(1)^5*T(4)-T(3)*T(4)+T(5)^2
↦ norid[5]=T(1)^6*T(3)-T(1)*T(3)^2+T(4)*T(5)
↦ norid[6]=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);
↦
↦ // 'normal' created a list of 3 ring(s).
↦ // nor[3+1] is the delta-invariant in case of choose=wd.
↦ // To see the rings, type (if the name of your list is nor):
↦      show( nor);
↦ // To access the 1-st ring and map (similar for the others), type:
↦      def R = nor[1]; setring R;  norid; normap;
↦ // R/norid is the 1-st ring of the normalization and
↦ // normap the map from the original basering to R/norid
  def S1=qr[1]; def S2=qr[2];
  setring S1; norid;
↦ norid[1]=0
  setring S2; norid;
↦ norid[1]=0
```

A.23 Branches of an Isolated Space Curve Singularity

In this example, the number of branches of a given quasihomogeneous isolated space curve singularity will be computed as an example of the pitfalls appearing in the use of primary decomposition. When dealing with singularities, two situations are possible in which the primary decomposition algorithm might not lead to a complete decomposition: first of all, one of the computed components could be globally irreducible, but analytically reducible (this is impossible for quasihomogeneous singularities) and, as a second possibility, a component might be irreducible over the rational numbers, but reducible over the complex numbers.

```
  ring r=0,(x,y,z),ds;
  ideal i=x^4-y*z^2,x*y-z^3,y^2-x^3*z;  // the space curve singularity
  qhweight(i);
↦ 1,2,1
  // The given space curve singularity is quasihomogeneous. Hence we can pass
  // to the polynomial ring.
  ring rr=0,(x,y,z),dp;
  ideal i=imap(r,i);
  resolution ires=mres(i,0);
  ires;
↦   1       3       2
↦ rr <--  rr <--  rr
↦
↦ 0       1       2
↦
  // From the structure of the resolution, we see that the Cohen-Macaulay
  // type of the given singularity is 2
  //
  // Let us now look for the branches using the primdec library.
  LIB "primdec.lib";
  primdecSY(i);
↦ [1]:
↦    [1]:
↦       _[1]=x-z
↦       _[2]=z2-y
↦    [2]:
↦       _[1]=x-z
↦       _[2]=z2-y
↦ [2]:
↦    [1]:
↦       _[1]=z3-xy
↦       _[2]=x3+x2z+xz2+xy+yz
↦       _[3]=x2z2+x2y+xyz+yz2+y2
↦    [2]:
↦       _[1]=z3-xy
↦       _[2]=x3+x2z+xz2+xy+yz
↦       _[3]=x2z2+x2y+xyz+yz2+y2
  def li=_[2];
  ideal i2=li[2];       // call the second ideal i2
  // The curve seems to have 2 branches by what we computed using the
  // algorithm of Shimoyama-Yokoyama.
  // Now the same computation by the Gianni-Trager-Zacharias algorithm:
  primdecGTZ(i);
↦ [1]:
↦    [1]:
↦       _[1]=-z2+y
↦       _[2]=x-z
↦    [2]:
↦       _[1]=-z2+y
↦       _[2]=x-z
```

```
↦ [2]:
↦    [1]:
↦       _[1]=z8+yz6+y2z4+y3z2+y4
↦       _[2]=xz5+z6+yz4+y2z2+y3
↦       _[3]=-z3+xy
↦       _[4]=x2z2+xz3+xyz+yz2+y2
↦       _[5]=x3+x2z+xz2+xy+yz
↦    [2]:
↦       _[1]=z8+yz6+y2z4+y3z2+y4
↦       _[2]=xz5+z6+yz4+y2z2+y3
↦       _[3]=-z3+xy
↦       _[4]=x2z2+xz3+xyz+yz2+y2
↦       _[5]=x3+x2z+xz2+xy+yz
  // Having computed the primary decomposition in 2 different ways and
  // having obtained the same number of branches, we might expect that the
  // number of branches is really 2, but we can check this by formulae
  // for the invariants of space curve singularities:
  //
  // mu = tau - t + 1 (for quasihomogeneous curve singularities)
  // where mu denotes the Milnor number, tau the Tjurina number and
  // t the Cohen-Macaulay type
  //
  // mu = 2 delta - r + 1
  // where delta denotes the delta-Invariant and r the number of branches
  //
  // tau can be computed by using the corresponding procedure T1 from
  // sing.lib.
  setring r;
  LIB "sing.lib";
  T_1(i);
↦ // dim T_1 = 13
↦ _[1]=gen(6)+2z*gen(5)
↦ _[2]=gen(4)+3x2*gen(2)
↦ _[3]=gen(3)+gen(1)
↦ _[4]=x*gen(5)-y*gen(2)-z*gen(1)
↦ _[5]=x*gen(1)-z2*gen(2)
↦ _[6]=y*gen(5)+3x2z*gen(2)
↦ _[7]=y*gen(2)-z*gen(1)
↦ _[8]=2y*gen(1)-z2*gen(5)
↦ _[9]=z2*gen(5)
↦ _[10]=z2*gen(1)
↦ _[11]=x3*gen(2)
↦ _[12]=x2z2*gen(2)
↦ _[13]=xz3*gen(2)
↦ _[14]=z4*gen(2)
  setring rr;
  // Hence tau is 13 and therefore mu is 12. But then it is impossible that
  // the singularity has two branches, since mu is even and delta is an
  // integer!
  // So obviously, we did not decompose completely. Because the first branch
  // is smooth, only the second ideal can be the one which can be decomposed
  // further.
  // Let us now consider the normalization of this second ideal i2.
```

```
LIB "normal.lib";
normal(i2);
↦
↦ // 'normal' created a list of 1 ring(s).
↦ // nor[1+1] is the delta-invariant in case of choose=wd.
↦ // To see the rings, type (if the name of your list is nor):
↦      show( nor);
↦ // To access the 1-st ring and map (similar for the others), type:
↦      def R = nor[1]; setring R;  norid; normap;
↦ // R/norid is the 1-st ring of the normalization and
↦ // normap the map from the original basering to R/norid
↦ [1]:
↦    //   characteristic : 0
↦ //   number of vars : 2
↦ //        block   1 : ordering a
↦ //                  : names    T(1) T(2)
↦ //                  : weights     1    0
↦ //        block   2 : ordering dp
↦ //                  : names    T(1) T(2)
↦ //        block   3 : ordering C
def rno=_[1];
setring rno;
norid;
↦ norid[1]=T(2)^4+T(2)^3+T(2)^2+T(2)+1
// The ideal is generated by a polynomial in one variable of degree 4 which
// factors completely into 4 polynomials of type T(2)+a.
// From this, we know that the ring of the normalization is the direct sum of
// 4 polynomial rings in one variable.
// Hence our original curve has these 4 branches plus a smooth one
// which we already determined by primary decomposition.
// Our final result is therefore: 5 branches.
```

A.24 Kernel of module homomorphisms

Let A , B be two matrices of size $m \times r$ and $m \times s$ over the ring R and consider the corresponding maps

$$R^r \xrightarrow{A} R^m \xleftarrow{B} R^s \ .$$

We want to compute the kernel of the map $R^r \xrightarrow{A} R^m \longrightarrow R^m/\mathrm{Im}(B)$. This can be done using the `modulo` command:

$$\texttt{modulo}(A,B) = \ker(R^r \xrightarrow{A} R^m/\mathrm{Im}(B)) \ .$$

```
ring r=0,(x,y,z),(c,dp);
matrix A[2][2]=x,y,z,1;
matrix B[2][2]=x2,y2,z2,xz;
print(modulo(A,B));
↦ yz2-x2, xyz-y2,  x2z-xy, x3-y2z,
↦ x2z-xz2,-x2z+y2z,xyz-yz2,0
```

A.25 Algebraic dependence

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

1. $f_1, \ldots, f_r$ are algebraically dependent.

   Let $I = \langle Y_1 - f_1, \ldots, Y_r - f_r \rangle \subseteq K[x_1, \ldots, x_n, Y_1, \ldots, Y_r]$. Then $I \cap K[Y_1, \ldots, Y_r]$ are the algebraic relations between $f_1, \ldots, f_r$.

2. $g \in K[f_1, \ldots, f_r]$.

   $g \in K[f_1, \ldots, f_r]$ if and only if the normal form of g with respect to I and a block ordering with respect to $X = (x_1, \ldots, x_n)$ and $Y = (Y_1, \ldots, Y_r)$ with $X > Y$ is in $K[Y]$.

Both questions can be answered using the following procedure. If the second argument is zero, it checks for algebraic dependence and returns the ideal of relations between the generators of the given ideal. Otherwise it checks for subring membership and returns the normal form of the second argument with respect to the ideal I.

```
proc algebraicDep(ideal J, poly g)
{
  def R=basering;         // give a name to the basering
  int n=size(J);
  int k=nvars(R);
  int i;
  intvec v;

  // construction of the new ring:

  // construct a weight vector
  v[n+k]=0;         // gives a zero vector of length n+k
  for(i=1;i<=k;i++)
  {
    v[i]=1;
  }
  string orde="(a("+string(v)+"),dp);";
  string ri="ring Rhelp=("+charstr(R)+"),
                      ("+varstr(R)+",Y(1.."+string(n)+")),"+orde;
                        // ring definition as a string
  execute(ri);            // execution of the string

  // construction of the new ideal I=(J[1]-Y(1),...,J[n]-Y(n))
  ideal I=imap(R,J);
  for(i=1;i<=n;i++)
  {
    I[i]=I[i]-var(k+i);
  }
  poly g=imap(R,g);
  if(g==0)
  {
    // construction of the ideal of relations by elimination
    poly el=var(1);
    for(i=2;i<=k;i++)
    {
      el=el*var(i);
    }
    ideal KK=eliminate(I,el);
    keepring(Rhelp);
    return(KK);
  }
```

```
  // reduction of g with respect to I
  ideal KK=reduce(g,std(I));
  keepring(Rhelp);
  return(KK);
}

// applications of the procedure
ring r=0,(x,y,z),dp;
ideal i=xz,yz;
algebraicDep(i,0);
↦ _[1]=0
// Note: after call of algebraicDep(), the basering is Rhelp.
setring r; kill Rhelp;
ideal j=xy+z2,z2+y2,x2y2-2xy3+y4;
algebraicDep(j,0);
↦ _[1]=Y(1)^2-2*Y(1)*Y(2)+Y(2)^2-Y(3)
setring r; kill Rhelp;
poly g=y2z2-xz;
algebraicDep(i,g);
↦ _[1]=Y(2)^2-Y(1)
// this shows that g is contained in i.
setring r; kill Rhelp;
algebraicDep(j,g);
↦ _[1]=-z^4+z^2*Y(2)-x*z
// this shows that g is contained in j.
```

A.26 Classification

Classification of isolated hypersurface singularities with respect to right equivalence is provided by the command `classify` of the library `classify.lib`. The classification is done using the algorithm of Arnold. Before entering this algorithm, a first guess based on the Hilbert polynomial of the Milnor algebra is made.

```
LIB "classify.lib";
ring r=0,(x,y,z),ds;
poly p=singularity("E[6k+2]",2)[1];
p=p+z^2;
p;
↦ z2+x3+xy6+y8
// We received an E_14 singularity in normal form
// from the database of normal forms. Since only the residual
// part is saved in the database, we added z^2 to get an E_14
// of embedding dimension 3.
//
// Now we apply a coordinate change in order to deal with a
// singularity which is not in normal form:
map phi=r,x+y,y+z,x;
poly q=phi(p);
// Yes, q really looks ugly, now:
q;
↦ x2+x3+3x2y+3xy2+y3+xy6+y7+6xy5z+6y6z+15xy4z2+15y5z2+20xy3z3+20y4z3+15xy2z\
  4+15y3z4+6xyz5+6y2z5+xz6+yz6+y8+8y7z+28y6z2+56y5z3+70y4z4+56y3z5+28y2z6+8\
  yz7+z8
```

```
  // Classification
  classify(q);
↦ About the singularity :
↦           Milnor number(f)   = 14
↦           Corank(f)          = 2
↦           Determinacy       <= 12
↦ Guessing type via Milnorcode:   E[6k+2]=E[14]
↦ 
↦ Computing normal form ...
↦ I have to apply the splitting lemma. This will take some time....:-)
↦    Arnold step number 9
↦ The singularity
↦    x3-9/4x4+27/4x5-189/8x6+737/8x7+6x6y+15x5y2+20x4y3+15x3y4+6x2y5+xy6-24\
  089/64x8-x7y+11/2x6y2+26x5y3+95/2x4y4+47x3y5+53/2x2y6+8xy7+y8+104535/64x9\
  +27x8y+135/2x7y2+90x6y3+135/2x5y4+27x4y5+9/2x3y6-940383/128x10-405/4x9y-2\
  025/8x8y2-675/2x7y3-2025/8x6y4-405/4x5y5-135/8x4y6+4359015/128x11+1701/4x\
  10y+8505/8x9y2+2835/2x8y3+8505/8x7y4+1701/4x6y5+567/8x5y6-82812341/512x12\
  -15333/8x11y-76809/16x10y2-25735/4x9y3-78525/16x8y4-16893/8x7y5-8799/16x6\
  y6-198x5y7-495/4x4y8-55x3y9-33/2x2y10-3xy11-1/4y12
↦ is R-equivalent to E[14].
↦    Milnor number = 14
↦    modality      = 1
↦ 2z2+x3+xy6+y8
  // The library also provides routines to determine the corank of q
  // and its residual part without going through the whole
  // classification algorithm.
  corank(q);
↦ 2
  morsesplit(q);
↦ y3-9/4y4+27/4y5-189/8y6+737/8y7+6y6z+15y5z2+20y4z3+15y3z4+6y2z5+yz6-24089\
  /64y8-y7z+11/2y6z2+26y5z3+95/2y4z4+47y3z5+53/2y2z6+8yz7+z8+104535/64y9+27\
  y8z+135/2y7z2+90y6z3+135/2y5z4+27y4z5+9/2y3z6-940383/128y10-405/4y9z-2025\
  /8y8z2-675/2y7z3-2025/8y6z4-405/4y5z5-135/8y4z6+4359015/128y11+1701/4y10z\
  +8505/8y9z2+2835/2y8z3+8505/8y7z4+1701/4y6z5+567/8y5z6-82812341/512y12-15\
  333/8y11z-76809/16y10z2-25735/4y9z3-78525/16y8z4-16893/8y7z5-8799/16y6z6-\
  198y5z7-495/4y4z8-55y3z9-33/2y2z10-3yz11-1/4z12
```

A.27 Fast lexicographical GB

Compute Groebner basis in lexicographical ordering by using the FGLM algorithm (`stdfglm`) and Hilbert driven Groebner (`stdhilb`).

The command `stdfglm` applies 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;
↦ 1
```

A.28 Parallelization with MPtcp links

In this example, we demonstrate how MPtcp links can be used to parallelize computations.

To compute a standard basis for a zero-dimensional ideal in the lexicographical ordering, one of the two powerful routines `stdhilb` (see Section 5.1.122 [stdhilb], page 204) and `stdfglm` (see Section 5.1.121 [stdfglm], page 204) should be used. However, a priory one can not predict which one of the two commands is faster. This very much depends on the (input) example. Therefore, we use MPtcp links to let both commands work on the problem independently and in parallel, so that the one which finishes first delivers the result.

The example we use is the so-called "omdi 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 explanatory 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 explicitly need to get the process id numbers of the competing SINGULAR processes, so that we can "kill" the looser later.
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 both child processes. Therefore, the innermost `eval` can be omitted (as is done for the `l_hilb` link), and only the identifier `i` needs to be communicated to the children. However, when `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 approximately once 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 has won delivers the result and is terminated with the usual `close` command. The other child which is still computing needs to be terminated by an explicit (i.e., system) kill command, 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 3.3 [Rings and orderings], page 28). 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 has 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.2.5 [Module orderings], page 277). However, in any case the procedure `show` from the library `inout.lib` displays the bracket format.

A vector $v = [f_1,...,f_n]$ should always be considered as a column vector in a free module of rank equal to nrows(v) where nrows(v) is equal to the maximal index r such that $f_r \neq 0$. This is due to the fact, that internally v is a polynomial in a sparse representation, i.e., $f_i * gen(i)$ is not stored if $f_i = 0$ (for reasons of efficiency), hence the last 0-entries of v are lost. Only more complex structures are able to keep the rank.

A module M in SINGULAR is given by a list of vectors $v_1,...,v_k$ which generate the module as a submodule of the free module of rank equal to nrows(M) which is the maximum of nrows(v_i).

If one wants to create a module with a larger rank than given by its generators, one has to use the command `attrib(M,"rank",r)` (see Section 5.1.1 [attrib], page 121, Section 5.1.86 [nrows], page 176) or to define a matrix first, then converting it into a module. Modules in SINGULAR are almost the same as matrices, they may be considered as sparse representations of matrices. A module of a matrix is generated by the columns of the matrix and a matrix of a module has as columns the generators of the module. These conversions preserve the rank and the number of generators, resp. the number of rows and columns.

By the above remarks it might appear that SINGULAR is only able to handle submodules of a free module. However, this is not true. SINGULAR can compute with any finitely generated module over the basering R. Such a module, say N, is not represented by its generators but by its (generators and) relations. This means that $N = R^n/M$ where n is the number of generators of N and $M \subseteq R^n$ is the module of relations. In other words, defining a module M as a submodule of a free module R^n can also be considered as the definition of $N = R^n/M$.

Note that most functions, when applied to a module M, really deal with M. However, there are some functions which deal with $N = R^n/M$ instead of M.

For example, `std(M)` computes a standard basis of M (and thus gives another representation of N as $N = R^n/\mathrm{std}(M)$). However, `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 module of M, i.e., the module of relations of the given generators of M which is equal to the second syzygy module of N. Refer to the description of each function in Section 5.1 [Functions], page 121 to get information which module the function deals with.

The numbering in `res` and other commands for computing resolutions refers to a resolution of $N = R^n/M$ (see Section 5.1.109 [res], page 193; Section C.3 [Syzygies and resolutions], page 282).

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 5.1.106 [reduce], page 191).

B.2 Monomial orderings

B.2.1 Introduction to orderings

SINGULAR offers a great variety of monomial orderings which provide an enormous functionality, if used diligently. However, this flexibility might also be confusing for the novice user. Therefore, we recommend to those not familiar with monomial orderings to generally use the ordering `dp` for computations in the polynomial ring $K[x_1, \ldots, x_n]$, resp. `ds` for computations in the localization $\mathrm{Loc}_{(x)}K[x_1, \ldots, x_n]$.

For inhomogeneous input ideals, standard (resp. groebner) bases computations are generally faster with the orderings $\mathrm{Wp}(w_1, \ldots, w_n)$ (resp. $\mathrm{Ws}(w_1, \ldots, w_n)$) if the input is quasihomogeneous w.r.t. the weights w_1, …, w_n of x_1, …, x_n.

If the output needs to be "triangular" (resp. "block-triangular"), the lexicographical ordering `lp` (resp. lexicographical block-orderings) need to be used. However, these orderings usually result in much less efficient computations.

B.2.2 General definitions for orderings

A monomial ordering (term ordering) on $K[x_1, \ldots, x_n]$ is a total ordering $<$ on the set of monomials (power products) $\{x^\alpha \mid \alpha \in \mathbf{N^n}\}$ which is compatible with the natural semigroup structure, i.e., $x^\alpha < x^\beta$ implies $x^\gamma x^\alpha < x^\gamma x^\beta$ for any $\gamma \in \mathbf{N^n}$. We do not require $<$ to be a well ordering. See the literature cited in Section C.7 [References], page 291.

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 well orderings (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 well orderings.

Let K be the ground field, $x = (x_1, \ldots, x_n)$ the variables and $<$ a monomial ordering, then Loc $K[x]$ denotes the localization of $K[x]$ with respect to the multiplicatively closed set

$$\{1 + g \mid g = 0 \text{ or } g \in K[x]\backslash\{0\} \text{ and } L(g) < 1\}.$$

Here, $L(g)$ denotes the leading monomial of g, i.e., the biggest monomial of g with respect to $<$. The result of any computation which uses standard basis computations has to be interpreted in Loc $K[x]$.

Note that the definition of a ring includes the definition of its monomial ordering (see Section 3.3 [Rings and orderings], page 28). SINGULAR offers the monomial orderings described in the following sections.

B.2.3 Global orderings

For all these orderings: Loc $K[x] = K[x]$

lp: lexicographical ordering:
$x^\alpha < x^\beta \Leftrightarrow \exists\, 1 \le i \le n : \alpha_1 = \beta_1, \ldots, \alpha_{i-1} = \beta_{i-1}, \alpha_i < \beta_i$.

rp: reverse lexicographical ordering:
$x^\alpha < x^\beta \Leftrightarrow \exists\, 1 \le i \le n : \alpha_n = \beta_n, \ldots, \alpha_{i+1} = \beta_{i+1}, \alpha_i > \beta_i$.

dp: degree reverse lexicographical ordering:
let $\deg(x^\alpha) = \alpha_1 + \cdots + \alpha_n$, then $x^\alpha < x^\beta \Leftrightarrow \deg(x^\alpha) < \deg(x^\beta)$ or
$\deg(x^\alpha) = \deg(x^\beta)$ and $\exists\, 1 \le i \le n : \alpha_n = \beta_n, \ldots, \alpha_{i+1} = \beta_{i+1}, \alpha_i > \beta_i$.

Dp: degree lexicographical ordering:
let $\deg(x^\alpha) = \alpha_1 + \cdots + \alpha_n$, then $x^\alpha < x^\beta \Leftrightarrow \deg(x^\alpha) < \deg(x^\beta)$ or
$\deg(x^\alpha) = \deg(x^\beta)$ and $\exists\, 1 \le i \le n : \alpha_1 = \beta_1, \ldots, \alpha_{i-1} = \beta_{i-1}, \alpha_i < \beta_i$.

wp: weighted reverse lexicographical ordering:
let $w_1, \ldots, w_n$ be positive integers. Then `wp`$(w_1, \ldots, w_n)$ is defined as `dp` but with $\deg(x^\alpha) = w_1\alpha_1 + \cdots + w_n\alpha_n$.

Wp: weighted lexicographical ordering:
let $w_1, \ldots, w_n$ be positive integers. Then `Wp`$(w_1, \ldots, w_n)$ is defined as `Dp` but with $\deg(x^\alpha) = w_1\alpha_1 + \cdots + w_n\alpha_n$.

B.2.4 Local orderings

For ls, ds, Ds and, if the weights are positive integers, also for ws and Ws, we have Loc $K[x] = K[x]_{(x)}$, the localization of $K[x]$ at the maximal ideal $(x_1, ..., x_n)$.

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:
let $\deg(x^\alpha) = \alpha_1 + \cdots + \alpha_n$, then $x^\alpha < x^\beta \Leftrightarrow \deg(x^\alpha) > \deg(x^\beta)$ or
$\deg(x^\alpha) = \deg(x^\beta)$ and $\exists\, 1 \le i \le n : \alpha_n = \beta_n, \ldots, \alpha_{i+1} = \beta_{i+1}, \alpha_i > \beta_i$.

Ds: negative degree lexicographical ordering:
let $\deg(x^\alpha) = \alpha_1 + \cdots + \alpha_n$, then $x^\alpha < x^\beta \Leftrightarrow \deg(x^\alpha) > \deg(x^\beta)$ or
$\deg(x^\alpha) = \deg(x^\beta)$ and $\exists\, 1 \le i \le n : \alpha_1 = \beta_1, \ldots, \alpha_{i-1} = \beta_{i-1}, \alpha_i < \beta_i$.

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.2.5 Module orderings

SINGULAR offers also orderings on the set of "monomials" $\{x^a e_i \mid a \in N^n, 1 \le i \le r\}$ in Loc $K[x]^r$ = Loc $K[x]e_1 + \ldots +$Loc $K[x]e_r$, where $e_1, \ldots, e_r$ denote the canonical generators of Loc $K[x]^r$, the r-fold direct sum of Loc $K[x]$. (The function `gen(i)` yields e_i).

We have two possibilities: either to give priority to the component of a vector in Loc $K[x]^r$ or (which is the default in SINGULAR) to give priority to the coefficients. The orderings `(<,c)` and `(<,C)` give priority to the coefficients; whereas `(c,<)` and `(C,<)` give priority to the components. Let `<` be any of the monomial orderings of Loc $K[x]$ as above.

(<,C): $<_m = (<, C)$ denotes the module ordering (giving priority to the coefficients):

$x^\alpha e_i <_m x^\beta e_j \Leftrightarrow x^\alpha < x^\beta$ or ($x^\alpha = x^\beta$ and $i < j$).

Example:

```
  ring r = 0, (x,y,z), ds;
  // the same as ring r = 0, (x,y,z), (ds, C);
  [x+y2,z3+xy];
↦ x*gen(1)+xy*gen(2)+y2*gen(1)+z3*gen(2)
  [x,x,x];
↦ x*gen(3)+x*gen(2)+x*gen(1)
```

(C,<): $<_m = (C, <)$ denotes the module ordering (giving priority to the component):

$x^\alpha e_i <_m x^\beta e_j \Leftrightarrow i < j$ or ($i = j$ and $x^\alpha < x^\beta$).

Example:

```
  ring r = 0, (x,y,z), (C,lp);
  [x+y2,z3+xy];
↦ xy*gen(2)+z3*gen(2)+x*gen(1)+y2*gen(1)
  [x,x,x];
↦ x*gen(3)+x*gen(2)+x*gen(1)
```

(<,c): $<_m = (<, c)$ denotes the module ordering (giving priority to the coefficients):

$x^\alpha e_i <_m x^\beta e_j \Leftrightarrow x^\alpha < x^\beta$ or ($x^\alpha = x^\beta$ and $i > j$).

Example:

```
  ring r = 0, (x,y,z), (lp,c);
  [x+y2,z3+xy];
↦ xy*gen(2)+x*gen(1)+y2*gen(1)+z3*gen(2)
  [x,x,x];
↦ x*gen(1)+x*gen(2)+x*gen(3)
```

(c,<): $<_m = (c, <)$ denotes the module ordering (giving priority to the component):

$x^\alpha e_i <_m x^\beta e_j \Leftrightarrow i > j$ or ($i = j$ and $x^\alpha < x^\beta$).

Example:

```
  ring r = 0, (x,y,z), (c,lp);
  [x+y2,z3+xy];
↦ [x+y2,xy+z3]
  [x,x,x];
↦ [x,x,x]
```

The output of a vector v in $K[x]^r$ with components $v_1, \ldots, v_r$ has the format $v_1 * gen(1) + \ldots + v_r * gen(r)$ (up to permutation) unless the ordering starts with `c`. In this case a vector is written as $[v_1, \ldots, v_r]$. In all cases SINGULAR can read input in both formats.

B.2.6 Matrix orderings

Let M be an invertible $(n \times n)$-matrix with integer coefficients and $M_1, \ldots, M_n$ the rows of M.

The M-ordering < is defined as follows:

$x^a < x^b \Leftrightarrow \exists\ 1 \leq i \leq n : M_1 a = M_1 b, \ldots, M_{i-1} a = M_{i-1} b$ and $M_i a < M_i b$.

Thus, $x^a < x^b$ if and only if Ma is smaller than Mb with respect to the lexicographical ordering.

The following matrices represent (for 3 variables) the global and local orderings defined above (note that the matrix is not uniquely determined by the ordering):

lp: $\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$ dp: $\begin{pmatrix} 1 & 1 & 1 \\ 0 & 0 & -1 \\ 0 & -1 & 0 \end{pmatrix}$ Dp: $\begin{pmatrix} 1 & 1 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix}$

wp(1,2,3): $\begin{pmatrix} 1 & 2 & 3 \\ 0 & 0 & -1 \\ 0 & -1 & 0 \end{pmatrix}$ Wp(1,2,3): $\begin{pmatrix} 1 & 2 & 3 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix}$

ls: $\begin{pmatrix} -1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & -1 \end{pmatrix}$ ds: $\begin{pmatrix} -1 & -1 & -1 \\ 0 & 0 & -1 \\ 0 & -1 & 0 \end{pmatrix}$ Ds: $\begin{pmatrix} -1 & -1 & -1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix}$

ws(1,2,3): $\begin{pmatrix} -1 & -2 & -3 \\ 0 & 0 & -1 \\ 0 & -1 & 0 \end{pmatrix}$ Ws(1,2,3): $\begin{pmatrix} -1 & -2 & -3 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix}$

Product orderings (see next section) represented by a matrix:

(dp(3), wp(1,2,3)): $\begin{pmatrix} 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & -1 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 2 & 3 \\ 0 & 0 & 0 & 0 & 0 & -1 \\ 0 & 0 & 0 & 0 & -1 & 0 \end{pmatrix}$

(Dp(3), ds(3)): $\begin{pmatrix} 1 & 1 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & -1 & -1 & -1 \\ 0 & 0 & 0 & 0 & 0 & -1 \\ 0 & 0 & 0 & 0 & -1 & 0 \end{pmatrix}$

Orderings with extra weight vector (see below) represented by a matrix:

(dp(3), a(1,2,3),dp(3)): $\begin{pmatrix} 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & -1 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 2 & 3 \\ 0 & 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & -1 \\ 0 & 0 & 0 & 0 & -1 & 0 \end{pmatrix}$

(a(1,2,3,4,5),Dp(3), ds(3)): $\begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 0 \\ 1 & 1 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & -1 & -1 & -1 \\ 0 & 0 & 0 & 0 & 0 & -1 \\ 0 & 0 & 0 & 0 & -1 & 0 \end{pmatrix}$

Example:

```
ring r = 0, (x,y,z), M(1, 0, 0,   0, 1, 0,   0, 0, 1);
```

which may also be written as:

```
  intmat m[3][3]=1, 0, 0, 0, 1, 0, 0, 0, 1;
  m;
↦ 1,0,0,
↦ 0,1,0,
↦ 0,0,1
  ring r = 0, (x,y,z), M(m);
  r;
↦ //   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 $n \times n$ 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.2.7 Product orderings

Let $x = (x_1, \ldots, x_n)$ and $y = (y_1, \ldots, y_m)$ be two ordered sets of variables, $<_1$ a monomial ordering on $K[x]$ and $<_2$ a monomial ordering on $K[y]$. The product ordering (or block ordering) $< := (<_1, <_2)$ on $K[x, y]$ is the following:
$x^a y^b < x^A y^B \Leftrightarrow x^a <_1 x^A$ or ($x^a = x^A$ and $y^b <_2 y^B$).

Inductively one defines the product ordering of more than two monomial orderings.

In SINGULAR, any of the above global orderings, local orderings or matrix orderings may be combined (in an arbitrary manner and length) to a product ordering. E.g., `(lp(3), M(1, 2, 3, 1, 1, 1, 1, 0, 0), ds(4), ws(1,2,3))` defines: `lp` on the first 3 variables, the matrix ordering `M(1, 2, 3, 1, 1, 1, 1, 0, 0)` on the next 3 variables, `ds` on the next 4 variables and `ws(1,2,3)` on the last 3 variables.

B.2.8 Extra weight vector

`a`$(w_1, \ldots, w_n)$, $w_1, \ldots, w_n$ any integers (including 0), defines $\deg(x^\alpha) = w_1\alpha_1 + \cdots + w_n\alpha_n$ and

$$\deg(x^\alpha) < \deg(x^\beta) \Rightarrow x^\alpha < x^\beta,$$
$$\deg(x^\alpha) > \deg(x^\beta) \Rightarrow x^\alpha > x^\beta.$$

An extra weight vector does not define a monomial ordering by itself: it can only be used in combination with other orderings to insert an extra line of weights into the ordering matrix.

Example:

```
ring r = 0, (x,y,z),  (a(1,2,3),wp(4,5,2));
ring s = 0, (x,y,z),  (a(1,2,3),dp);
ring q = 0, (a,b,c,d),(lp(1),a(1,2,3),ds);
```

Appendix C Mathematical background

This chapter introduces some of the mathematical notions and definitions used throughout the manual. It is mostly a collection of the most prominent definitions and properties. For details, please, refer to some articles or text books (see Section C.7 [References], page 291).

C.1 Standard bases

Definition

Let $R = \mathrm{Loc}_< K[\underline{x}]$ and let I be a submodule of R^r. Note that for r=1 this means that I is an ideal in R. Denote by $L(I)$ the submodule of R^r generated by the leading terms of elements of I, i.e. by $\{L(f) \mid f \in I\}$. Then $f_1, \ldots, f_s \in I$ is called a **standard basis** of I if $L(f_1), \ldots, L(f_s)$ generate $L(I)$.

Properties

normal form:

A function $\mathrm{NF} : R^r \times \{G \mid G \text{ a standard basis}\} \to R^r, (p, G) \mapsto \mathrm{NF}(p|G)$, is called a **normal form** if for any $p \in R^r$ and any standard basis G the following holds: if $\mathrm{NF}(p|G) \neq 0$ then $L(g)$ does not divide $L(\mathrm{NF}(p|G))$ for all $g \in G$.

$\mathrm{NF}(p|G)$ is called a **normal form of** p **with respect to** G (note that such a function is not unique).

ideal membership:

For a standard basis G of I the following holds: $f \in I$ if and only if $\mathrm{NF}(f, G) = 0$.

Hilbert function:

Let $I \subseteq K[\underline{x}]^r$ be a homogeneous module, then the Hilbert function H_I of I (see below) and the Hilbert function $H_{L(I)}$ of the leading module $L(I)$ coincide, i.e., $H_I = H_{L(I)}$.

C.2 Hilbert function

Let $\mathrm{M} = \bigoplus_i M_i$ be a graded module over $K[x_1, .., x_n]$ with respect to weights $(w_1, ..w_n)$. The **Hilbert function** of M, H_M, is defined (on the integers) by

$$H_M(k) := dim_K M_k.$$

The **Hilbert-Poincare series** of M is the power series

$$\mathrm{HP}_M(t) := \sum_{i=-\infty}^{\infty} H_M(i) t^i = \sum_{i=-\infty}^{\infty} dim_K M_i \cdot t^i.$$

It turns out that $\mathrm{HP}_M(t)$ can be written in two useful ways for weights $(1, .., 1)$:

$$\mathrm{HP}_M(t) = \frac{Q(t)}{(1-t)^n} = \frac{P(t)}{(1-t)^{dim(M)}}$$

where $Q(t)$ and $P(t)$ are polynomials in $\mathbf{Z}[t]$. $Q(t)$ is called the **first Hilbert series**, and $P(t)$ the **second Hilbert series**. If $P(t) = \sum_{k=0}^N a_k t^k$, and $d = dim(M)$, then $H_M(s) = \sum_{k=0}^N a_k \binom{d+s-k-1}{d-1}$ (the **Hilbert polynomial**) for $s \geq N$.

Generalizing these to quasihomogeneous modules we get

$$\mathrm{HP}_M(t) = \frac{Q(t)}{\Pi_{i=1}^n (1 - t^{w_i})}$$

where $Q(t)$ is a polynomial in $\mathbf{Z}[t]$. $Q(t)$ is called the **first (weighted) Hilbert series** of M.

C.3 Syzygies and resolutions

Syzygies

Let R be a quotient of $\mathrm{Loc}_< K[\underline{x}]$ and let $I=(g_1,...,g_s)$ be a submodule of R^r. Then the **module of syzygies** (or **1st syzygy module**, **module of relations**) of I, $\mathrm{syz}(I)$, is defined to be the kernel of the map $R^s \rightarrow R^r$, $\sum_{i=1}^s w_i e_i \mapsto \sum_{i=1}^s w_i g_i$.

The **k-th syzygy module** is defined inductively to be the module of syzygies of the $(k-1)$-st syzygy module.

Note, that the syzygy modules of I depend on a choice of generators $g_1,...,g_s$. But one can show that they depend on I uniquely up to direct summands.

Example:

```
  ring R= 0,(u,v,x,y,z),dp;
  ideal i=ux, vx, uy, vy;
  print(syz(i));
↦ -y,0, -v,0,
↦ 0, -y,u, 0,
↦ x, 0, 0, -v,
↦ 0, x, 0, u
```

Free resolutions

Let $I=(g_1,...,g_s)\subseteq R^r$ and $M=R^r/I$. A **free resolution of** M is a long exact sequence

$$...\longrightarrow F_2 \xrightarrow{A_2} F_1 \xrightarrow{A_1} F_0 \longrightarrow M \longrightarrow 0,$$

where the columns of the matrix A_1 generate I . Note, that resolutions need not to be finite (i.e., of finite length). The Hilbert Syzygy Theorem states, that for $R=\mathrm{Loc}_< K[\underline{x}]$ there exists a ("minimal") resolution of length not exceeding the number of variables.

Example:

```
  ring R= 0,(u,v,x,y,z),dp;
  ideal I = ux, vx, uy, vy;
  resolution resI = mres(I,0); resI;
↦  1      4      4      1
↦ R <--  R <--  R <--  R
↦
↦ 0      1      2      3
↦
  // The matrix A_1 is given by
  print(matrix(resI[1]));
↦ vy,uy,vx,ux
  // We see that the columns of A_1 generate I.
  // The matrix A_2 is given by
  print(matrix(resI[3]));
↦ u,
↦ -v,
↦ -x,
↦ y
```

Betti numbers and regularity

Let R be a graded ring (e.g., $R = \mathrm{Loc}_< K[\underline{x}]$) and let $I \subset R^r$ be a graded submodule. Let

$$R^r = \bigoplus_a R \cdot e_{a,0} \xleftarrow{A_1} \bigoplus_a R \cdot e_{a,1} \longleftarrow \ldots \longleftarrow \bigoplus_a R \cdot e_{a,n} \longleftarrow 0$$

be a minimal free resolution of R^n/I considered with homogeneous maps of degree 0. Then the **graded Betti number** $b_{i,j}$ of R^r/I is the minimal number of generators $e_{a,j}$ in degree $i+j$ of the j-th syzygy module of R^r/I (i.e., the $(j-1)$-st syzygy module of I). Note, that by definition the 0-th syzygy module of R^r/I is R^r and the 1st syzygy module of R^r/I is I.

The **regularity** of I is the smallest integer s

such that

$$\deg(e_{a,j}) \le s + j - 1 \quad \text{for all } j.$$

Example:

```
  ring R= 0,(u,v,x,y,z),dp;
  ideal I = ux, vx, uy, vy;
  resolution resI = mres(I,0); resI;
↦  1      4      4      1
↦ R <--  R <--  R <--  R
↦
↦ 0      1      2      3
↦
  // the betti number:
  print(betti(resI), "betti");
↦            0     1     2     3
↦ ------------------------------
↦     0:     1     -     -     -
↦     1:     -     4     4     1
↦ ------------------------------
↦ total:     1     4     4     1
  // the regularity:
  regularity(resI);
↦ 2
```

C.4 Characteristic sets

Let $<$ be the lexicographical ordering on $R = K[x_1, ..., x_n]$ with $x_1 < ... < x_n$. For $f \in R$ let $\mathrm{lvar}(f)$ (the leading variable of f) be the largest variable in f, i.e., if $f = a_s(x_1, ..., x_{k-1})x_k^s + ... + a_0(x_1, ..., x_{k-1})$ for some $k \le n$ then $\mathrm{lvar}(f) = x_k$.

Moreover, let $\mathrm{ini}(f) := a_s(x_1, ..., x_{k-1})$. The pseudo remainder $r = \mathrm{prem}(g, f)$ of g with respect to f is defined by the equality $\mathrm{ini}(f)^a \cdot g = qf + r$ with $\deg_{lvar(f)}(r) < \deg_{lvar(f)}(f)$ and a minimal.

A set $T = \{f_1, ..., f_r\} \subset R$ is called triangular if $\mathrm{lvar}(f_1) < ... < \mathrm{lvar}(f_r)$. Moreover, let $U \subset T$, then (T, U) is called a triangular system, if T is a triangular set such that $\mathrm{ini}(T)$ does not vanish on $V(T) \setminus V(U) (=: V(T \setminus U))$.

T is called irreducible if for every i there are no d_i, f_i', f_i'' such that

$$\mathrm{lvar}(d_i) < \mathrm{lvar}(f_i) = \mathrm{lvar}(f_i') = \mathrm{lvar}(f_i''),$$

$$0 \notin \mathrm{prem}(\{d_i, \mathrm{ini}(f_i'), \mathrm{ini}(f_i'')\}, \{f_1, ..., f_{i-1}\}),$$

$$\mathrm{prem}(d_i f_i - f_i' f_i'', \{f_1, ..., f_{i-1}\}) = 0.$$

Furthermore, (T, U) is called irreducible if T is irreducible.

The main result on triangular sets is the following: let $G = \{g_1, ..., g_s\} \subset R$ then there are irreducible triangular sets $T_1, ..., T_l$ such that $V(G) = \bigcup_{i=1}^{l}(V(T_i \setminus I_i))$ where $I_i = \{\mathrm{ini}(f) \mid f \in T_i\}$. Such a set $\{T_1, ..., T_l\}$ is called an **irreducible characteristic series** of the ideal (G).

Example:

```
  ring R= 0,(x,y,z,u),dp;
  ideal i=-3zu+y2-2x+2,
          -3x2u-4yz-6xz+2y2+3xy,
          -3z2u-xu+y2z+y;
  print(char_series(i));
↦ _[1,1],3x2z-y2+2yz,3x2u-3xy-2y2+2yu,
↦ x,     -y+2z,      -2y2+3yu-4
```

C.5 Gauss-Manin connection

Let $f: (C^{n+1}, 0) \to (C, 0)$ be a complex isolated hypersurface singularity given by a polynomial with algebraic coefficients which we also denote by f. Let $O = C[x_0, \ldots, x_n]_{(x_0,\ldots,x_n)}$ be the local ring at the origin and J_f the Jacobian ideal of f.

A **Milnor representative** of f defines a differentiable fibre bundle over the punctured disc with fibres of homotopy type of μ n-spheres. The n-th cohomology bundle is a flat vector bundle of dimension n and carries a natural flat connection with covariant derivative ∂_t. The **monodromy operator** is the action of a positively oriented generator of the fundamental group of the puctured disc on the Milnor fibre. Sections in the cohomology bundle of **moderate growth** at 0 form a regular $D = C\{t\}[\partial_t]$-module G, the **Gauss-Manin connection**.

By integrating along flat multivalued families of cycles, one can consider fibrewise global holomorphic differential forms as elements of G. This factors through an inclusion of the **Brieskorn lattice** $H'' := \Omega^{n+1}_{C^{n+1},0}/df \wedge d\Omega^{n-1}_{C^{n+1},0}$ in G.

The D-module structure defines the **V-filtration** V on G by $V^\alpha := \sum_{\beta \geq \alpha} C\{t\} ker(t\partial_t - \beta)^{n+1}$. The Brieskorn lattice defines the **Hodge filtration** F on G by $F_k = \partial_t^k H''$ which comes from the **mixed Hodge structure** on the Milnor fibre. Note that $F_{-1} = H'$.

The induced V-filtration on the Brieskorn lattice determines the **singularity spectrum** Sp by $Sp(\alpha) := \dim_C Gr_V^\alpha Gr_0^F G$. The spectrum consists of μ rational numbers $\alpha_1, \ldots, \alpha_\mu$ such that $e^{2\pi i \alpha_1}, \ldots, e^{2\pi i \alpha_\mu}$ are the eigenvalues of the monodromy. These **spectral numbers** lie in the open interval $(-1, n)$, symmetric about the midpoint $(n-1)/2$.

The spectrum is constant under μ-constant deformations and has the following semicontinuity property: The number of spectral numbers in an interval $(a, a+1]$ of all singularities of a small deformation of f is greater or equal to that of f in this interval. For semiquasihomogeneous singularities, this also holds for intervals of the form $(a, a+1)$.

Two given isolated singularities f and g determine two spectra and from these spectra we get an integer. This integer is the maximal positive integer k such that the semicontinuity holds for the spectrum of f and k times the spectrum of g. These numbers give bounds for the maximal number of isolated singularities of a specific type on a hypersurface $X \subset P^n$ of degree d: such a hypersurface has a smooth hyperplane section, and the complement is a small deformation of a cone over this hyperplane section. The cone itself being a μ-constant deformation of $x_0^d + \ldots + x_n^d = 0$, the singularities are bounded by the spectrum of $x_0^d + \ldots + x_n^d$.

Using the library `gaussman.lib` one can compute the **monodromy**, the V-filtration on H''/H', and the spectrum.

Let us consider as an example $f = x^5 + x^2y^2 + y^5$. First, we compute a matrix M such that $\exp(2\pi iM)$ is a monodromy matrix of f and the Jordan normal form of M :

```
LIB "gaussman.lib";
ring R=0,(x,y),ds;
poly f=x5+x2y2+y5;
list l=monodromy(f);
matrix M=jordanmatrix(l[1],l[2],l[3]);
print(M);
↦ 1/2,0,  0,    0,    0,    0,    0,0,    0,    0,    0,
↦ 1,  1/2,0,    0,    0,    0,    0,0,    0,    0,    0,
↦ 0,  0,  7/10,0,    0,    0,    0,0,    0,    0,    0,
↦ 0,  0,  0,    7/10,0,    0,    0,0,    0,    0,    0,
↦ 0,  0,  0,    0,    9/10,0,    0,0,    0,    0,    0,
↦ 0,  0,  0,    0,    0,    9/10,0,0,    0,    0,    0,
↦ 0,  0,  0,    0,    0,    0,    1,0,    0,    0,    0,
↦ 0,  0,  0,    0,    0,    0,    0,11/10,0,    0,    0,
↦ 0,  0,  0,    0,    0,    0,    0,0,    11/10,0,    0,
↦ 0,  0,  0,    0,    0,    0,    0,0,    0,    13/10,0,
↦ 0,  0,  0,    0,    0,    0,    0,0,    0,    0,    13/10
```

Now, we compute the V-filtration on H''/H' and the spectrum:

```
LIB "gaussman.lib";
ring R=0,(x,y),ds;
poly f=x5+x2y2+y5;
list l=vfilt(f);
print(l[1]);
↦ -1/2,
↦ -3/10,
↦ -1/10,
↦ 0,
↦ 1/10,
↦ 3/10,
↦ 1/2
print(l[2]);
↦ 1,2,2,1,2,2,1
print(l[3]);
↦ [1]:
↦    _[1]=gen(11)
↦ [2]:
↦    _[1]=gen(10)
↦    _[2]=gen(6)
↦ [3]:
↦    _[1]=gen(9)
↦    _[2]=gen(4)
↦ [4]:
↦    _[1]=gen(5)
↦ [5]:
↦    _[1]=gen(3)
↦    _[2]=gen(8)
↦ [6]:
↦    _[1]=gen(2)
↦    _[2]=gen(7)
↦ [7]:
```

```
↦     _[1]=gen(1)
  print(l[4]);
↦ y5,
↦ y4,
↦ y3,
↦ y2,
↦ xy,
↦ y,
↦ x4,
↦ x3,
↦ x2,
↦ x,
↦ 1
```

Here `l[1]` contains the spectral numbers, `l[2]` the corresponding multiplicities, `l[3]` a C -basis of the V-filtration on H''/H' in terms of the monomial basis of $O/J_f \cong H''/H'$ in `l[4]`.

If the principal part of f is C-nondegenerate, one can compute the spectrum using the library `spectrum.lib`. In this case, the V-filtration on H'' coincides with the Newton-filtration on H'' which allows to compute the spectrum more efficiently.

Let us calculate one specific example, the maximal number of triple points of type $\tilde{E}_6$ on a surface $X \subset P^3$ of degree seven. This calculation can be done over the rationals. So choose a local ordering on $Q[x,y,z]$. Here we take the negative degree lexicographical ordering which is denoted `ds` in SINGULAR:

```
ring r=0,(x,y,z),ds;
LIB "spectrum.lib";
poly f=x^7+y^7+z^7;
list s1=spectrumnd( f );
s1;
↦ [1]:
↦     _[1]=-4/7
↦     _[2]=-3/7
↦     _[3]=-2/7
↦     _[4]=-1/7
↦     _[5]=0
↦     _[6]=1/7
↦     _[7]=2/7
↦     _[8]=3/7
↦     _[9]=4/7
↦     _[10]=5/7
↦     _[11]=6/7
↦     _[12]=1
↦     _[13]=8/7
↦     _[14]=9/7
↦     _[15]=10/7
↦     _[16]=11/7
↦ [2]:
↦     1,3,6,10,15,21,25,27,27,25,21,15,10,6,3,1
```

The command `spectrumnd(f)` computes the spectrum of f and returns a list with six entries: The Milnor number $\mu(f)$, the geometric genus $p_g(f)$ and the number of different spectrum numbers. The other three entries are of type `intvec`. They contain the numerators, denominators and multiplicities of the spectrum numbers. So $x^7+y^7+z^7=0$ has Milnor number 216 and geometrical genus 35. Its spectrum consists of the 16 different rationals

$\frac{3}{7}, \frac{4}{7}, \frac{5}{7}, \frac{6}{7}, \frac{1}{1}, \frac{8}{7}, \frac{9}{7}, \frac{10}{7}, \frac{11}{7}, \frac{12}{7}, \frac{13}{7}, \frac{2}{1}, \frac{15}{7}, \frac{16}{7}, \frac{17}{7}, \frac{18}{7}$
appearing with multiplicities
1,3,6,10,15,21,25,27,27,25,21,15,10,6,3,1.

The singularities of type $\tilde{E}_6$ form a μ-constant one parameter family given by $x^3+y^3+z^3+\lambda xyz = 0, \quad \lambda^3 \neq -27$. Therefore they have all the same spectrum, which we compute for $x^3+y^3+z^3$.

```
poly g=x^3+y^3+z^3;
list s2=spectrumnd(g);
s2;
↦ [1]:
↦    8
↦ [2]:
↦    1
↦ [3]:
↦    4
↦ [4]:
↦    1,4,5,2
↦ [5]:
↦    1,3,3,1
↦ [6]:
↦    1,3,3,1
```

Evaluating semicontinuity is very easy:

```
semicont(s1,s2);
↦ 18
```

This tells us that there are at most 18 singularities of type $\tilde{E}_6$ on a septic in P^3. But $x^7+y^7+z^7$ is semiquasihomogeneous (sqh), so we can also apply the stronger form of semicontinuity:

```
semicontsqh(s1,s2);
↦ 17
```

So in fact a septic has at most 17 triple points of type $\tilde{E}_6$.

Note that `spectrumnd(f)` works only if f has nondegenerate principal part. In fact `spectrumnd` will detect a degenerate principal part in many cases and print out an error message. However if it is known in advance that f has nondegenerate principal part, then the spectrum may be computed much faster using `spectrumnd(f,1)`.

C.6 Toric ideals and integer programming

C.6.1 Toric ideals

Let A denote an $m \times n$ matrix with integral coefficients. For $u \in \mathbb{Z}^n$, we define u^+, u^- to be the uniquely determined vectors with nonnegative coefficients and disjoint support (i.e., $u_i^+ = 0$ or $u_i^- = 0$ for each component i) such that $u = u^+ - u^-$. For $u \geq 0$ component-wise, let x^u denote the monomial $x_1^{u_1} \cdot \ldots \cdot x_n^{u_n} \in K[x_1, \ldots, x_n]$.

The ideal

$$I_A := < x^{u^+} - x^{u^-} | u \in \ker(A) \cap \mathbb{Z}^n > \subset K[x_1, \ldots, x_n]$$

is called a **toric ideal.**

The first problem in computing toric ideals is to find a finite generating set: Let $v_1, \ldots, v_r$ be a lattice basis of $\ker(A) \cap \mathbb{Z}^n$ (i.e, a basis of the $\mathbb{Z}$-module). Then

$$I_A := I : (x_1 \cdot \ldots \cdot x_n)^\infty$$

where

$$I = < x^{v_i^+} - x^{v_i^-} | i = 1, \ldots, r >$$

The required lattice basis can be computed using the LLL-algorithm (see [[Coh93]], page 290). For the computation of the saturation, there are various possibilities described in the section Algorithms.

C.6.2 Algorithms

The following algorithms are implemented in Section D.4.11 [toric_lib], page 424.

C.6.2.1 The algorithm of Conti and Traverso

The algorithm of Conti and Traverso (see [[CoTr91]], page 290) computes I_A via the extended matrix $B = (I_m|A)$, where I_m is the $m \times m$ unity matrix. A lattice basis of B is given by the set of vectors $(a^j, -e_j) \in \mathbb{Z}^{m+n}$, where a^j is the j-th row of A and e_j the j-th coordinate vector. We look at the ideal in $K[y_1, \ldots, y_m, x_1, \ldots, x_n]$ corresponding to these vectors, namely

$$I_1 = < y^{a_j^+} - x_j y^{a_j^-} | j = 1, \ldots, n > .$$

We introduce a further variable t and adjoin the binomial $t \cdot y_1 \cdot \ldots \cdot y_m - 1$ to the generating set of I_1, obtaining an ideal I_2 in the polynomial ring $K[t, y_1, \ldots, y_m, x_1, \ldots, x_n]$. I_2 is saturated w.r.t. all variables because all variables are invertible modulo I_2. Now I_A can be computed from I_2 by eliminating the variables $t, y_1, \ldots, y_m$.

Because of the big number of auxiliary variables needed to compute a toric ideal, this algorithm is rather slow in practice. However, it has a special importance in the application to integer programming (see Section C.6.4 [Integer programming], page 290).

C.6.2.2 The algorithm of Pottier

The algorithm of Pottier (see [[Pot94]], page 291) starts by computing a lattice basis $v_1, \ldots, v_r$ for the integer kernel of A using the LLL-algorithm. The ideal corresponding to the lattice basis vectors

$$I_1 = < x^{v_i^+} - x^{v_i^-} | i = 1, \ldots, r >$$

is saturated – as in the algorithm of Conti and Traverso – by inversion of all variables: One adds an auxiliary variable t and the generator $t \cdot x_1 \cdot \ldots \cdot x_n - 1$ to obtain an ideal I_2 in $K[t, x_1, \ldots, x_n]$ from which one computes I_A by elimination of t.

C.6.2.3 The algorithm of Hosten and Sturmfels

The algorithm of Hosten and Sturmfels (see [[HoSt95]], page 291) allows to compute I_A without any auxiliary variables, provided that A contains a vector w with positive coefficients in its row space. This is a real restriction, i.e., the algorithm will not necessarily work in the general case.

A lattice basis $v_1, \ldots, v_r$ is again computed via the LLL-algorithm. The saturation step is performed in the following way: First note that w induces a positive grading w.r.t. which the ideal

$$I = < x^{v_i^+} - x^{v_i^-} | i = 1, \ldots, r >$$

corresponding to our lattice basis is homogeneous. We use the following lemma:

Let I be a homogeneous ideal w.r.t. the weighted reverse lexicographical ordering with weight vector w and variable order $x_1 > x_2 > \ldots > x_n$. Let G denote a Groebner basis of I w.r.t. to this ordering. Then a Groebner basis of $(I : x_n^\infty)$ is obtained by dividing each element of G by the highest possible power of x_n.

From this fact, we can successively compute

$$I_A = I : (x_1 \cdot \ldots \cdot x_n)^\infty = (((I : x_1^\infty) : x_2^\infty) : \ldots : x_n^\infty);$$

in the i-th step we take x_i as the cheapest variable and apply the lemma with x_i instead of x_n.

This procedure involves n Groebner basis computations. Actually, this number can be reduced to at most $n/2$ (see [[HoSh98]], page 291), and the single computations – except from the first one – show to be easy and fast in practice.

C.6.2.4 The algorithm of Di Biase and Urbanke

Like the algorithm of Hosten and Sturmfels, the algorithm of Di Biase and Urbanke (see [[DBUr95]], page 291) performs up to $n/2$ Groebner basis computations. It needs no auxiliary variables, but a supplementary precondition; namely, the existence of a vector without zero components in the kernel of A.

The main idea comes from the following observation:

Let B be an integer matrix, $u_1, \ldots, u_r$ a lattice basis of the integer kernel of B. Assume that all components of u_1 are positive. Then

$$I_B = < x^{u_i^+} - x^{u_i^-} | i = 1, \ldots, r >,$$

i.e., the ideal on the right is already saturated w.r.t. all variables.

The algorithm starts by finding a lattice basis $v_1, \ldots, v_r$ of the kernel of A such that v_1 has no zero component. Let $\{i_1, \ldots, i_l\}$ be the set of indices i with $v_{1,i} < 0$. Multiplying the components $i_1, \ldots, i_l$ of $v_1, \ldots, v_r$ and the columns $i_1, \ldots, i_l$ of A by -1 yields a matrix B and a lattice basis $u_1, \ldots, u_r$ of the kernel of B that fulfill the assumption of the observation above. We are then able to compute a generating set of I_A by applying the following "variable flip" successively to $i = i_1, \ldots, i_l$:

Let $>$ be an elimination ordering for x_i. Let A_i be the matrix obtained by multiplying the i-th column of A with -1. Let

$$\{x_i^{r_j} x^{a_j} - x^{b_j} | j \in J\}$$

be a Groebner basis of I_{A_i} w.r.t. $>$ (where x_i is neither involved in x^{a_j} nor in x^{b_j}). Then

$$\{x^{a_j} - x_i^{r_j} x^{b_j} | j \in J\}$$

is a generating set for I_A.

C.6.2.5 The algorithm of Bigatti, La Scala and Robbiano

The algorithm of Bigatti, La Scala and Robbiano (see [[BLR98]], page 290) combines the ideas of the algorithms of Pottier and of Hosten and Sturmfels. The computations are performed on a graded ideal with one auxiliary variable u and one supplementary generator $x_1 \cdot \ldots \cdot x_n - u$ (instead of the generator $t \cdot x_1 \cdot \ldots \cdot x_n - 1$ in the algorithm of Pottier). The algorithm uses a quite unusual technique to get rid of the variable u again.

There is another algorithm of the authors which tries to parallelize the computations (but which is not implemented in this library).

C.6.3 The Buchberger algorithm for toric ideals

Toric ideals have a very special structure that allows us to improve the Buchberger algorithm in many respects: They are prime ideals and generated by binomials. Pottier used this fact to describe all operations of the Buchberger algorithm on the ideal generators in terms of vector additions and subtractions. Some other strategies like multiple reduction (see [[CoTr91]], page 290) or the use of bit vectors to represent the support of a monomial (see [[Big97]], page 290) may be applied to more general ideals, but show to be especially useful in the toric case.

C.6.4 Integer programming

Let A be an $m \times n$ matrix with integral coefficients, $b \in \mathbb{Z}^m$ and $c \in \mathbb{Z}^n$. The problem

$$\min\{c^T x | x \in \mathbb{Z}^n, Ax = b, x \geq 0 \text{ component-wise}\}$$

is called an instance of the **integer programming problem** or **IP problem.**

The IP problem is very hard; namely, it is NP-complete.

For the following discussion let $c \geq 0$ (component-wise). We consider c as a weight vector; because of its non-negativity, c can be refined into a monomial ordering $>_c$. It turns out that we can solve such an IP instance with the help of toric ideals:

First we assume that an initial solution v (i.e., $v \in \mathbb{Z}^n, v \geq 0, Av = b$) is already known. We obtain the optimal solution v_0 (i.e., with $c^T v_0$ minimal) by the following procedure:

- (1) Compute the toric ideal I(A) using one of the algorithms in the previous section.
- (2) Compute the reduced Groebner basis G(c) of I(A) w.r.t. $>_c$.
- (3) Reduce x^v modulo G(c) using the Hironaka division algorithm. If the result of this reduction is $x^(v_0)$, then v_0 is an optimal solution of the given instance.

If no initial solution is known, we are nevertheless able to solve the problem with similar techniques. For this purpose we replace our instance by an extended instance with the matrix used in the Conti-Traverso algorithm. Indeed, the Conti-Traverso algorithm offers the possibility to verify solvability of a given instance and to find an initial solution in the case of existence (but none of the other algorithms does!). Details can be found in [CoTr91] and [The99].

An implementation of the above algorithm and some examples can be found in Section D.4.10 [intprog_lib], page 422.

Classical methods for solving IP instances like Branch-and-Bound methods seem to be faster in general than the methods using toric ideals. But the latter have one great advantage: If one wants to solve various instances that differ only by the vector b , one has to perform steps (1) and (2) above only once. As the running time of step (3) is very short, solving all the instances is not much harder than solving one single instance.

For a detailed discussion see [The99].

C.6.5 Relevant References

- [Big97] Bigatti, A.M.: Computation of Hilbert-Poincare series. Journal of Pure and Applied Algebra (1997) 199, 237-253
- [BLR98] Bigatti, A.M.; La Scala, R.; Robbiano, L.: Computing toric ideals. Journal of Symbolic Computation (to appear)
- [Coh93] Cohen, H.: A Course in Computational Algebraic Number Theory. Springer (1997)
- [CoTr91] Conti, P.; Traverso, C.: Buchberger algorithm and integer programming. Proceedings AAECC-9 (new Orleans), Springer LNCS (1991) 539, 130-139

- [DBUr95] Di Biase, F.; Urbanke, R.: An algorithm to calculate the kernel of certain polynomial ring homomorphisms. Experimental Mathematics (1995) 4, 227-234
- [HoSh98] Hosten, S.; Shapiro, J.: Primary decomposition of lattice basis ideals. (to appear)
- [HoSt95] Hosten, S.; Sturmfels, B.: GRIN: An implementation of Groebner bases for integer programming. in Balas, E.; Clausen, J. (editors): Integer Programming and Combinatorial Optimization. Springer LNCS (1995) 920, 267-276
- [Pot94] Pottier, L.: Groebner bases of toric ideals. Rapport de recherche 2224 (1997), INRIA Sophia Antipolis
- [Stu96] Sturmfels, B.: Groebner Bases and Convex Polytopes. University Lecture Series, Volume 8 (1996), American Mathematical Society
- [The99] Theis, C.: Der Buchberger-Algorithmus fuer torische Ideale und seine Anwendung in der ganzzahligen Optimierung. Diplomarbeit, Universitaet des Saarlandes (1999), Saarbruecken (Germany)

C.7 References

The Centre for Computer Algebra Kaiserslautern publishes a series of preprints which are electronically available at `http://www.mathematik.uni-kl.de/~zca/Reports_on_ca`. Other sources to check are `http://symbolicnet.mcs.kent.edu/`, `http://www.can.nl/`,... and the following list of books:

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.; Greuel, G.-M.; Pfister, G.: Primary decomposition: algorithms and comparisons. Preprint, Univ. Kaiserslautern, 1998. To appear in: Greuel, G.-M.; Matzat, B. H.; Hiss, G. (Eds.), Algorithmic Algebra and Number Theory. Springer Verlag, Heidelberg, 1998
- Decker, W.; Greuel, G.-M.; de Jong, T.; Pfister, G.: The normalization: a new algorithm, implementation and comparisons. Preprint, Univ. Kaiserslautern, 1998
- Decker, W.; Heydtmann, A.; Schreyer, F. O.: Generating a Noetherian Normalization of the Invariant Ring of a Finite Group, 1997, to appear in Journal of Symbolic Computation

- Faugère, J. C.; Gianni, P.; Lazard, D.; Mora, T.: Efficient computation of zero-dimensional Gröbner bases by change of ordering. Journal of Symbolic Computation, 1989
- Gräbe, H.-G.: On factorized Gröbner bases, Univ. Leipzig, Inst. für Informatik, 1994
- Grassmann, H.; Greuel, G.-M.; Martin, B.; Neumann, W.; Pfister, G.; Pohl, W.; Schönemann, H.; Siebert, T.: On an implementation of standard bases and syzygies in Singular. Proceedings of the Workshop Computational Methods in Lie theory in AAECC (1995)
- Greuel, G.-M.; Pfister, G.: Advances and improvements in the theory of standard bases and syzygies. Arch. d. Math. 63(1995)
- Kemper; Generating Invariant Rings of Finite Groups over Arbitrary Fields. 1996, to appear in Journal of Symbolic Computation
- Kemper and Steel: Some Algorithms in Invariant Theory of Finite Groups. 1997
- Lee, H.R.; Saunders, B.D.: Fraction Free Gaussian Elimination for Sparse Matrices. Journal of Symbolic Computation (1995) 19, 393-402
- Schönemann, H.: Algorithms in Singular, Reports on Computer Algebra 2(1996), Kaiserslautern
- Siebert, T.: On strategies and implementations for computations of free resolutions. Reports on Computer Algebra 8(1996), Kaiserslautern
- Wang, D.: Characteristic Sets and Zero Structure of Polynomial Sets. Lecture Notes, RISC Linz, 1989

Appendix D SINGULAR libraries

SINGULAR comes with a set of standard libraries. Their content is described in the following subsections.

Use the Section 5.1.65 [LIB], page 162 command for loading of single libraries, and the command `LIB "all.lib";` for loading all libraries.

D.1 standard_lib

The library `standard.lib` provides extensions to the set of built-in commands and is automatically loaded during the start of SINGULAR, unless SINGULAR is started up with the `--no-stdlib` command line option (see Section 3.1.6 [Command line options], page 18).

Library: standard.lib

Purpose: Procedures which are always loaded at Start-up

Procedures:

D.2 General purpose

D.2.1 all_lib

The library `all.lib` provides a convenient way to load all libraries of the SINGULAR distribution.

Example:

```
option(loadLib);
LIB "all.lib";
↦ // ** loaded all.lib (1.35.2.2,2001/05/17)
↦ // ** loaded makedbm.lib (1.11,2000/12/22)
↦ // ** loaded brnoeth.lib (1.11.2.4,2002/04/11)
↦ // ** loaded paramet.lib (1.11,2001/02/19)
↦ // ** loaded surf.lib (1.19.2.4,2002/02/18)
↦ // ** loaded latex.lib (1.19.2.1,2002/02/20)
↦ // ** loaded graphics.lib (1.10,2001/02/19)
↦ // ** loaded zeroset.lib (1.7.2.2,2002/02/20)
↦ // ** loaded ntsolve.lib (1.12,2001/02/19)
↦ // ** loaded triang.lib (1.7,2001/02/19)
↦ // ** loaded solve.lib (1.21.2.10,2002/03/12)
↦ // ** loaded presolve.lib (1.17.2.4,2002/03/06)
↦ // ** loaded stratify.lib (1.7.2.4,2002/04/11)
↦ // ** loaded rinvar.lib (1.7.2.3,2002/02/20)
↦ // ** loaded finvar.lib (1.32.2.1,2002/02/20)
↦ // ** loaded ainvar.lib (1.6.2.1,2002/02/20)
↦ // ** loaded spectrum.lib (1.12.2.3,2002/03/06)
↦ // ** loaded spcurve.lib (1.15.2.1,2002/02/20)
↦ // ** loaded sing.lib (1.24.2.3,2002/02/20)
↦ // ** loaded qhmoduli.lib (1.0,2000/12/12)
↦ // ** loaded mondromy.lib (1.22.2.2,2002/02/20)
↦ // ** loaded hnoether.lib (1.29.2.7,2002/03/08)
↦ // ** loaded gaussman.lib (1.33.2.23,2002/03/26)
↦ // ** loaded equising.lib (1.7,2001/02/05)
↦ // ** loaded deform.lib (1.25.2.1,2002/02/20)
```

```
↦ // ** loaded classify.lib (1.48.2.4,2002/04/11)
↦ // ** loaded toric.lib (1.11,2001/02/06)
↦ // ** loaded intprog.lib (1.5,2001/02/06)
↦ // ** loaded reesclos.lib (1.50,2001/08/06)
↦ // ** loaded primitiv.lib (1.15,2001/02/05)
↦ // ** loaded primdec.lib (1.98.2.10,2002/03/25)
↦ // ** loaded normal.lib (1.34.2.16,2002/03/08)
↦ // ** loaded mregular.lib (1.6.2.1,2002/02/20)
↦ // ** loaded mprimdec.lib (1.1.2.3,2002/03/19)
↦ // ** loaded homolog.lib (1.15.2.1,2002/02/20)
↦ // ** loaded elim.lib (1.14.2.2,2002/02/20)
↦ // ** loaded algebra.lib (1.9.2.2,2002/02/20)
↦ // ** redefining abs **
↦ // ** loaded linalg.lib (1.10.2.11,2002/03/26)
↦ // ** loaded matrix.lib (1.26.2.1,2002/02/20)
↦ // ** loaded ring.lib (1.17.2.1,2002/02/20)
↦ // ** loaded random.lib (1.16.2.1,2002/02/20)
↦ // ** loaded poly.lib (1.33.2.5,2002/04/09)
↦ // ** loaded inout.lib (1.21.2.3,2002/02/20)
↦ // ** loaded general.lib (1.38.2.6,2002/03/06)
```

D.2.2 general_lib

Library: general.lib

Purpose: Elementary Computations of General Type

Procedures:

D.2.2.1 A_Z

Procedure from library `general.lib` (see Section D.2.2 [general_lib], page 294).

Usage: A_Z("a",n); a any letter, n integer (-26<= n <=26, !=0)

Return: string of n small (if a is small) or capital (if a is capital) letters, comma separated, beginning with a, in alphabetical order (or revers alphabetical order if n<0)

Example:

```
LIB "general.lib";
A_Z("c",5);
↦ c,d,e,f,g
A_Z("Z",-5);
↦ Z,Y,X,W,V
string sR = "ring R = (0,"+A_Z("A",6)+"),("+A_Z("a",10)+"),dp;";
sR;
↦ ring R = (0,A,B,C,D,E,F),(a,b,c,d,e,f,g,h,i,j),dp;
execute(sR);
R;
↦ //   characteristic : 0
↦ //   6 parameter    : A B C D E F
↦ //   minpoly        : 0
↦ //   number of vars : 10
↦ //        block   1 : ordering dp
↦ //                  : names    a b c d e f g h i j
↦ //        block   2 : ordering C
```

D.2.2.2 ASCII

Procedure from library `general.lib` (see Section D.2.2 [general_lib], page 294).

Usage: ASCII([n,m]); n,m= integers (32 <= n <= m <= 126)

Return: string of printable ASCII characters (no native language support) ASCII(): string of all ASCII characters with its numbers, ASCII(n): n-th ASCII character
ASCII(n,m): n-th up to m-th ASCII character (inclusive)

Example:

```
LIB "general.lib";
ASCII();"";
↦      !    "    #    $    %    &    '    (    )    *    +    ,    -    .
↦ 32   33   34   35   36   37   38   39   40   41   42   43   44   45   46
↦ /    0    1    2    3    4    5    6    7    8    9    :    ;    <    =
↦ 47   48   49   50   51   52   53   54   55   56   57   58   59   60   61
↦ >    ?    @    A    B    C    D    E    F    G    H    I    J    K    L
↦ 62   63   64   65   66   67   68   69   70   71   72   73   74   75   76
↦ M    N    O    P    Q    R    S    T    U    V    W    X    Y    Z    [
↦ 77   78   79   80   81   82   83   84   85   86   87   88   89   90   91
↦ \    ]    ^    _    `    a    b    c    d    e    f    g    h    i    j
↦ 92   93   94   95   96   97   98   99  100  101  102  103  104  105  10
↦ k    l    m    n    o    p    q    r    s    t    u    v    w    x    y
↦ 107  108  109  110  111  112  113  114  115  116  117  118  119  120  121
↦ z    {    |    }    ~
↦ 122  123  124  125  126
↦
ASCII(42);
↦ *
ASCII(32,126);
↦  !"#$%&'()*+,-./0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefgh\
   ijklmnopqrstuvwxyz{|}~
```

D.2.2.3 absValue

Procedure from library `general.lib` (see Section D.2.2 [general_lib], page 294).

Usage: absValue(c); c int, number or poly

Return: absValue(c); the absolute value of c

Note: absValue(c)=c if c>=0; absValue=-c if c<=0.
So the function can be applied to any type, for which comparison
operators are defined.

Example:

```
LIB "general.lib";
ring r1 = 0,x,dp;
absValue(-2002);
↦ 2002
poly f=-4;
absValue(f);
↦ 4
```

See also: Section 4.3.5 [boolean expressions], page 76.

D.2.2.4 binomial

Procedure from library `general.lib` (see Section D.2.2 [general_lib], page 294).

Usage: binomial(n,k[,p]); n,k,p integers

Return: binomial(n,k); binomial coefficient n choose k
- of type string (computed in characteristic 0)
binomial(n,k,p); n choose k, computed in characteristic 0 or prime(p)
- of type number if a basering, say R, is present and p=0=char(R) or if prime(p)=char(R)
- of type string else

Note: In any characteristic, binomial(n,k) = coefficient of x^k in (1+x)^n

Example:

```
LIB "general.lib";
binomial(200,100);"";                  //type string, computed in char 0
↦ 90548514656103281165404177077484163874504589675413336841320
↦
binomial(200,100,3);"";                //type string, computed in char 3
↦ 0
↦
int n,k = 200,100;
ring r = 0,x,dp;
number b1 = binomial(n,k,0);           //type number, computed in ring r
poly b2 = coeffs((x+1)^n,x)[k+1,1];    //coefficient of x^k in (x+1)^n
b1-b2;                                 //b1 and b2 should coincide
↦ 0
```

See also: Section 5.1.96 [prime], page 183.

D.2.2.5 deleteSublist

Procedure from library `general.lib` (see Section D.2.2 [general_lib], page 294).

Usage: deleteSublist(v,l); intvec v; list l
where the entries of the integer vector v correspond to the positions of the elements to be deleted

Return: list without the deleted elements

Example:

```
LIB "general.lib";
list l=1,2,3,4,5;
intvec v=1,3,4;
l=deleteSublist(v,l);
l;
↦ [1]:
↦    2
↦ [2]:
↦    5
```

D.2.2.6 factorial

Procedure from library `general.lib` (see Section D.2.2 [general_lib], page 294).

Usage: factorial(n[,p]); n,p integers

Return: factorial(n): n! (computed in characteristic 0), of type string.
factorial(n,p): n! computed in characteristic 0 or prime(p)
- of type number if a basering is present and 0=p=char(basering) or if prime(p)=char(basering)
- of type string else

Example:

```
LIB "general.lib";
factorial(37);"";                 //37! of type string (as long integer)
↦ 13763753091226345046315979581580902400000000
↦
ring r1 = 0,x,dp;
number p = factorial(37,0);       //37! of type number, computed in r
p;
↦ 13763753091226345046315979581580902400000000
```

See also: Section 5.1.96 [prime], page 183.

D.2.2.7 fibonacci

Procedure from library `general.lib` (see Section D.2.2 [general_lib], page 294).

Usage: fibonacci(n); n,p integers

Return: fibonacci(n): nth Fibonacci number, f(0)=f(1)=1, f(i+1)=f(i-1)+f(i)
- computed in characteristic 0, of type string
fibonacci(n,p): f(n) computed in characteristic 0 or prime(p)
- of type number if a basering is present and p=0=char(basering) or if prime(p)=char(basering)
- of type string else

Example:

```
LIB "general.lib";
fibonacci(42); "";             //f(42) of type string (as long integer)
↦ 267914296
↦
ring r = 2,x,dp;
number b = fibonacci(42,2);    //f(42) of type number, computed in r
b;
↦ 0
```

See also: Section 5.1.96 [prime], page 183.

D.2.2.8 kmemory

Procedure from library `general.lib` (see Section D.2.2 [general_lib], page 294).

Usage: kmemory([n,[v]]); n,v integers

Return: memory in kilobyte of type int
n=0: memory used by active variables (same as no parameters)
n=1: total memory allocated by Singular
n=2: difference between top and init memory adress (sbrk memory)
n!=0,1,2: 0

Display: detailed information about allocated and used memory if v!=0

Note: kmemory uses internal function 'memory' to compute kilobyte, and is the same as 'memory' for n!=0,1,2

Example:

```
LIB "general.lib";
kmemory();
↦ 152
kmemory(1,1);
↦ // total memory allocated, at the moment, by SINGULAR (kilobyte):
↦ 652
```

D.2.2.9 killall

Procedure from library `general.lib` (see Section D.2.2 [general_lib], page 294).

Usage: killall(); (no parameter)
killall("type_name");
killall("not", "type_name");

Return: killall(); kills all user-defined variables except loaded procedures, no return value.
- killall("type_name"); kills all user-defined variables, of type "type_name"
- killall("not", "type_name"); kills all user-defined variables, except those of type "type_name" and except loaded procedures
- killall("not", "name_1", "name_2", ...); kills all user-defined variables, except those of name "name_i" and except loaded procedures

Note: killall should never be used inside a procedure

Example:

```
LIB "general.lib";
ring rtest; ideal i=x,y,z; string str="hi"; int j = 3;
export rtest,i,str,j;       //this makes the local variables global
↦ // ** `rtest` is already global
↦ // ** `i` is already global
↦ // ** `str` is already global
↦ // ** `j` is already global
listvar();
↦ // j                    [0]  int 3
↦ // str                  [0]  string hi
↦ // rtest                [0]  *ring
↦ //      i                    [0]  ideal, 3 generator(s)
↦ // LIB                  [0]  string standard.lib,general..., 74 char(s)
killall("ring");            // kills all rings
↦ // ** killing the basering for level 0
listvar();
↦ // j                    [0]  int 3
↦ // str                  [0]  string hi
↦ // LIB                  [0]  string standard.lib,general..., 74 char(s)
killall("not", "int");      // kills all variables except int's (and procs)
listvar();
↦ // j                    [0]  int 3
↦ // LIB                  [0]  string standard.lib,general..., 74 char(s)
killall();                  // kills all vars except loaded procs
```

```
listvar();
↦ // j                    [0]  int 3
↦ // LIB                  [0]  string standard.lib,general..., 74 char(s)
```

D.2.2.10 number_e

Procedure from library `general.lib` (see Section D.2.2 [general_lib], page 294).

Usage: number_e(n); n integer

Return: Euler number e=exp(1) up to n decimal digits (no rounding)
- of type string if no basering of char 0 is defined
- of type number if a basering of char 0 is defined

Display: decimal format of e if printlevel > 0 (default:printlevel=0)

Note: procedure uses algorithm of A.H.J. Sale

Example:

```
LIB "general.lib";
number_e(30);"";
↦ 2.718281828459045235360287471352
↦
ring R = 0,t,lp;
number e = number_e(30);
e;
↦ 13591409142295226176801437356763/500000000000000000000000000000
```

D.2.2.11 number_pi

Procedure from library `general.lib` (see Section D.2.2 [general_lib], page 294).

Usage: number_pi(n); n positive integer

Return: pi (area of unit circle) up to n decimal digits (no rounding)
- of type string if no basering of char 0 is defined,
- of type number, if a basering of char 0 is defined

Display: decimal format of pi if printlevel > 0 (default:printlevel=0)

Note: procedure uses algorithm of S. Rabinowitz

Example:

```
LIB "general.lib";
number_pi(11);"";
↦ 3.14159265358
↦
ring r = (real,10),t,dp;
number pi = number_pi(11); pi;
↦ 3.1415926536
```

D.2.2.12 primes

Procedure from library `general.lib` (see Section D.2.2 [general_lib], page 294).

Usage: primes(n,m); n,m integers

Return: intvec, consisting of all primes p, prime(n)<=p<=m, in increasing order if n<=m, resp. prime(m)<=p<=n, in decreasing order if m<n.

Note: prime(n); returns the biggest prime number <= min(n,32003) if n>=2, else 2

Example:

```
LIB "general.lib";
primes(50,100);"";
↦ 47,53,59,61,67,71,73,79,83,89,97
↦
intvec v = primes(37,1); v;
↦ 37,31,29,23,19,17,13,11,7,5,3,2
```

D.2.2.13 product

Procedure from library `general.lib` (see Section D.2.2 [general_lib], page 294).

Usage: product(id[,v]); id ideal/vector/module/matrix/intvec/intmat/list, v intvec (default: v=1..number of entries of id)

Assume: list members can be multiplied.

Return: The product of all entries of id [with index given by v] of type depending on the entries of id.

Note: If id is not a list, id is treated as a list of polys resp. integers. A module m is identified with the corresponding matrix M (columns of M generate m).
If v is outside the range of id, we have the empty product and the result will be 1 (of type int).

Example:

```
LIB "general.lib";
ring r= 0,(x,y,z),dp;
ideal m = maxideal(1);
product(m);
↦ xyz
product(m[2..3]);
↦ yz
matrix M[2][3] = 1,x,2,y,3,z;
product(M);
↦ 6xyz
intvec v=2,4,6;
product(M,v);
↦ xyz
intvec iv = 1,2,3,4,5,6,7,8,9;
v=1..5,7,9;
product(iv,v);
↦ 7560
intmat A[2][3] = 1,1,1,2,2,2;
product(A,3..5);
↦ 4
```

D.2.2.14 sort

Procedure from library `general.lib` (see Section D.2.2 [general_lib], page 294).

Usage: sort(id[v,o,n]); id = ideal/module/intvec/list(of intvec's or int's)
sort may be called with 1, 2 or 3 arguments in the following way:
sort(id[v,n]); v=intvec of positive integers, n=integer,
sort(id[o,n]); o=string (any allowed ordstr of a ring), n=integer

Return: a list l of two elements:

```
l[1]: object of same type as input but sorted in the following way:
  - if id=ideal/module: generators of id are sorted w.r.t. intvec v
    (id[v[1]] becomes 1-st, id[v[2]]  2-nd element, etc.). If no v is
    present, id is sorted w.r.t. ordering o (if o is given) or w.r.t.
    actual monomial ordering (if no o is given):
    NOTE: generators with SMALLER(!) leading term come FIRST
    (e.g. sort(id); sorts backwards to actual monomial ordering)
  - if id=list of intvec's or int's: consider a list element, say
    id[1]=3,2,5, as exponent vector of the monomial x^3*y^2*z^5;
    the corresponding monomials are ordered w.r.t. intvec v (s.a.).
    If no v is present, the monomials are sorted w.r.t. ordering o
    (if o is given) or w.r.t. lexicographical ordering (if no o is
    given). The corresponding ordered list of exponent vectors is
    returned.
    (e.g. sort(id); sorts lexicographically, smaller int's come first)
    WARNING: Since negative exponents create the 0 polynomial in
    Singular, id should not contain negative integers: the result
    might not be as expected
  - if id=intvec: id is treated as list of integers
  - if n!=0 the ordering is inverse, i.e. w.r.t. v(size(v)..1)
    default: n=0
 l[2]: intvec, describing the permutation of the input (hence l[2]=v
    if v is given (with positive integers))
```

Note: If v is given id may be any simply indexed object (e.g. any list or string); if v[i]<0 and i<=size(id) v[i] is set internally to i; entries of v must be pairwise distinct to get a permutation if id. Zero generators of ideal/module are deleted

Example:

```
LIB "general.lib";
ring r0 = 0,(x,y,z,t),lp;
ideal i = x3,z3,xyz;
sort(i);            //sorts using lex ordering, smaller polys come first
↦ [1]:
↦    _[1]=z3
↦    _[2]=xyz
↦    _[3]=x3
↦ [2]:
↦    2,3,1
sort(i,3..1);
↦ [1]:
↦    _[1]=xyz
↦    _[2]=z3
↦    _[3]=x3
↦ [2]:
↦    3,2,1
sort(i,"ls")[1];     //sort w.r.t. negative lex ordering
↦ _[1]=x3
↦ _[2]=xyz
↦ _[3]=z3
intvec v =1,10..5,2..4;v;
↦ 1,10,9,8,7,6,5,2,3,4
sort(v)[1];          // sort v lexicographically
```

```
↦ 1,2,3,4,5,6,7,8,9,10
sort(v,"Dp",1)[1];   // sort v w.r.t (total sum, reverse lex)
↦ 10,9,8,7,6,5,4,3,2,1
```

D.2.2.15 sum

Procedure from library `general.lib` (see Section D.2.2 [general_lib], page 294).

Usage: sum(id[,v]); id ideal/vector/module/matrix/intvec/intmat/list, v intvec (default: v=1..number of entries of id)

Assume: list members can be added.

Return: The sum of all entries of id [with index given by v] of type depending on the entries of id.

Note: If id is not a list, id is treated as a list of polys resp. integers. A module m is identified with the corresponding matrix M (columns of M generate m).
If v is outside the range of id, we have the empty sum and the result will be 0 (of type int).

Example:

```
LIB "general.lib";
ring r= 0,(x,y,z),dp;
vector pv = [xy,xz,yz,x2,y2,z2];
sum(pv);
↦ x2+xy+y2+xz+yz+z2
sum(pv,2..5);
↦ x2+y2+xz+yz
matrix M[2][3] = 1,x,2,y,3,z;
intvec w=2,4,6;
sum(M,w);
↦ x+y+z
intvec iv = 1,2,3,4,5,6,7,8,9;
sum(iv,2..4);
↦ 9
```

D.2.2.16 watchdog

Procedure from library `general.lib` (see Section D.2.2 [general_lib], page 294).

Return: Result of cmd, if the result can be computed in i seconds. Otherwise the computation is interrupted after i seconds, the string "Killed" is returned and the global variable 'watchdog_interrupt' is defined.

Note: * the MP package must be enabled
* the current basering should not be watchdog_rneu, since watchdog_rneu will be killed
* if there are variable names of the structure x(i) all polynomials have to be put into eval(...) in order to be interpreted correctly
* a second Singular process is started by this procedure

Example:

```
LIB "general.lib";
ring r=0,(x,y,z),dp;
poly f=x^30+y^30;
watchdog(1,"factorize(eval("+string(f)+"))");
```

```
↦ Killed
watchdog(100,"factorize(eval("+string(f)+"))");
↦ [1]:
↦    _[1]=1
↦    _[2]=x2+y2
↦    _[3]=x4-x2y2+y4
↦    _[4]=x8-x6y2+x4y4-x2y6+y8
↦    _[5]=x16+x14y2-x10y6-x8y8-x6y10+x2y14+y16
↦ [2]:
↦    1,1,1,1,1
```

D.2.2.17 which

Procedure from library `general.lib` (see Section D.2.2 [general_lib], page 294).

Usage: which(command); command = string expression

Return: Absolute pathname of command, if found in search path. Empty string, otherwise.

Note: Based on the Unix command 'which'.

Example:

```
LIB "general.lib";
which("sh");
↦ /bin/sh
```

D.2.2.18 primecoeffs

Procedure from library `general.lib` (see Section D.2.2 [general_lib], page 294).

Usage: primecoeffs(J[,q]); J any type which can be converted to a matrix e.g. ideal, matrix, vector, module, int, intvec
q = intger

Compute: primefactors <= min(p,32003) of coeffs of J (default p = 32003)

Return: a list, say l, of two intvectors:
l[1] : the different primefactors of all coefficients of J l[2] : the different remaining factors

Note: the procedure works for small integers only, just by testing all primes (not to be considerd as serious prime factorization!)

Example:

```
LIB "general.lib";
primecoeffs(intvec(7*8*121,7*8));"";
↦ [1]:
↦    2,7,11
↦ [2]:
↦    1
↦
ring r = 0,(b,c,t),dp;
ideal I = -13b6c3t+4b5c4t,-10b4c2t-5b4ct2;
primecoeffs(I);
↦ [1]:
↦    2,5,13
↦ [2]:
↦    _[1]=1
```

D.2.2.19 primefactors

Procedure from library `general.lib` (see Section D.2.2 [general_lib], page 294).

Usage: primefactors(n [,p]); n = int or number, p = integer

Compute: primefactors <= min(p,32003) of n (default p = 32003)

Return: a list, say l,
l[1] : primefactors <= min(p,32003) of n
l[2] : l[2][i] = multiplicity of l[1][i]
l[3] : remaining factor (n=product{ (l[1][i]^l[2][i])*l[3]}) type(l[3])=typeof(n)

Note: If n is a long integer (of type number) then the procedure finds primefactors <= min(p,32003) but n may be larger as 2147483647 (max. integer representation)

Warning: the procedure works for small integers only, just by testing all primes (not to be considerd as serious prime factorization!)

Example:

```
LIB "general.lib";
primefactors(7*8*121);
↦ [1]:
↦    2,7,11
↦ [2]:
↦    3,1,2
↦ [3]:
↦    1
ring r = 0,x,dp;
primefactors(123456789100);
↦ [1]:
↦    2,5
↦ [2]:
↦    2,2
↦ [3]:
↦    1234567891
```

D.2.3 inout_lib

Library: inout.lib

Purpose: Printing and Manipulating In- and Output

Procedures:

D.2.3.1 allprint

Procedure from library `inout.lib` (see Section D.2.3 [inout_lib], page 304).

Usage: allprint(L); L list

Display: prints L[1], L[2], ... if an integer with name ALLprint is defined.
makes "pause", if ALLprint > 0
listvar(matrix), if ALLprint = 2

Return: no return value

Example:

```
LIB "inout.lib";
ring S;
matrix M=matrix(freemodule(2),3,3);
int ALLprint; export ALLprint;
↦ // ** `ALLprint` is already global
allprint("M =",M);
↦ M =
↦ 1,0,0,
↦ 0,1,0,
↦ 0,0,0
kill ALLprint;
```

D.2.3.2 lprint

Procedure from library `inout.lib` (see Section D.2.3 [inout_lib], page 304).

Usage: lprint(id[,n]); id poly/ideal/vector/module/matrix, n integer

Return: string of id in a format fitting into lines of size n, such that no monomial is destroyed, i.e. the new line starts with + or -; (default: n = pagewidth).

Note: id is printed columnwise, each column separated by a blank line; hence lprint(transpose(id)); displays a matrix id in a format which can be used as input.

Example:

```
LIB "inout.lib";
ring r= 0,(x,y,z),ds;
poly f=((x+y)*(x-y)*(x+z)*(y+z)^2);
lprint(f,40);
↦   x3y2-xy4+2x3yz+x2y2z-2xy3z-y4z+x3z2
↦ +2x2yz2-xy2z2-2y3z2+x2z3-y2z3
module m = [f*(x-y)],[0,f*(x-y)];
string s=lprint(m); s;"";
↦   x4y2-x3y3-x2y4+xy5+2x4yz-x3y2z-3x2y3z+xy4z+y5z+x4z2+x3yz2-3x2y2z2-xy3z2
↦ +2y4z2+x3z3-x2yz3-xy2z3+y3z3,
↦   0,
↦ 
↦   0,
↦   x4y2-x3y3-x2y4+xy5+2x4yz-x3y2z-3x2y3z+xy4z+y5z+x4z2+x3yz2-3x2y2z2-xy3z2
↦ +2y4z2+x3z3-x2yz3-xy2z3+y3z3
↦ 
execute("matrix M[2][2]="+s+";");      //use the string s as input
module m1 = transpose(M);           //should be the same as m
print(m-m1);
↦ 0,0,
↦ 0,0
```

D.2.3.3 pmat

Procedure from library `inout.lib` (see Section D.2.3 [inout_lib], page 304).

Usage: pmat(M,[n]); M matrix, n integer

Display: display M in array format if it fits into pagewidth; if n is given, only the first n characters of each column are shown

Return: no return value

Example:

```
LIB "inout.lib";
ring r=0,(x,y,z),ls;
ideal i= x,z+3y,x+y,z;
matrix m[3][3]=i^2;
pmat(m);
↦ x2,     xz+3xy,     xy+x2,
↦ xz,     z2+6yz+9y2, yz+3y2+xz+3xy,
↦ z2+3yz, y2+2xy+x2,  yz+xz
pmat(m,3);
↦ x2  xz+ xy+
↦ xz  z2+ yz+
↦ z2+ y2+ yz+
```

D.2.3.4 rMacaulay

Procedure from library `inout.lib` (see Section D.2.3 [inout_lib], page 304).

Usage: rMacaulay(s[,n]); s string, n integer

Return: A string which should be readable by Singular if s is a string which was produced by Macaulay. If a second argument is present the first n lines of the file are deleted (which is useful if the file was produced e.g. by the putstd command of Macaulay).

Note: This does not always work with 'cut and paste' since the character \ is treated differently

Example:

```
LIB "inout.lib";
// Assume there exists a file 'Macid' with the following ideal in
// Macaulay format:"
// x[0]3-101/74x[0]2x[1]+7371x[0]x[1]2-13/83x[1]3-x[0]2x[2] \
//     -4/71x[0]x[1]x[2]
// Read this file into Singular and assign it to the string s1 by:
// string s1 = read("Macid");
// This is equivalent to";
string s1 =
"x[0]3-101/74x[0]2x[1]+7371x[0]x[1]2-13/83x[1]3-x[0]2x[2]-4/71x[0]x[1]x[2]";
rMacaulay(s1);
↦ x(0)^3-101/74*x(0)^2*x(1)+7371*x(0)*x(1)^2-13/83*x(1)^3-x(0)^2*x(2)-4/71*\
   x(0)*x(1)*x(2)
// You may wish to assign s1 to a Singular ideal id:
string sid = "ideal id =",rMacaulay(s1),";";
ring r = 0,x(0..3),dp;
execute(sid);
id; "";
↦ id[1]=x(0)^3-101/74*x(0)^2*x(1)+7371*x(0)*x(1)^2-13/83*x(1)^3-x(0)^2*x(2)\
   -4/71*x(0)*x(1)*x(2)
↦ 
// Now treat a matrix in Macaulay format. Using the execute
// command, this could be assinged to a Singular matrix as above.
string s2 = "
0  0  0  0  0
```

```
a3 0  0  0  0
0  b3 0  0  0
0  0  c3 0  0
0  0  0  d3 0
0  0  0  0  e3 ";
rMacaulay(s2);
↦ 0, 0, 0, 0, 0,
↦ a3,0, 0, 0, 0,
↦ 0, b3,0, 0, 0,
↦ 0, 0, c3,0, 0,
↦ 0, 0, 0, d3,0,
↦ 0, 0, 0, 0, e3
```

D.2.3.5 show

Procedure from library `inout.lib` (see Section D.2.3 [inout_lib], page 304).

Usage: show(id); id any object of basering or of type ring/qring
show(R,s); R=ring, s=string (s = name of an object belonging to R)

Display: display id/s in a compact format together with some information

Return: no return value

Note: objects of type string, int, intvec, intmat belong to any ring. id may be a ring or a qring. In this case the minimal polynomial is displayed, and, for a qring, also the defining ideal.
id may be of type list but the list must not contain a ring.
show(R,s) does not work inside a procedure!

Example:

```
LIB "inout.lib";
ring r;
show(r);
↦ // ring: (32003),(x,y,z),(dp(3),C);
↦ // minpoly = 0
↦ // objects belonging to this ring:
ideal i=x^3+y^5-6*z^3,xy,x3-y2;
show(i,3);            // introduce 3 space tabs before information
↦    // ideal, 3 generator(s)
↦ y5+x3-6z3,
↦ xy,
↦ x3-y2
vector v=x*gen(1)+y*gen(3);
module m=v,2*v+gen(4);
list L = i,v,m;
show(L);
↦ // list, 3 element(s):
↦ [1]:
↦    // ideal, 3 generator(s)
↦ y5+x3-6z3,
↦ xy,
↦ x3-y2
↦ [2]:
↦    // vector
```

```
↦ [x,0,y]
↦ [3]:
↦    // module, 2 generator(s)
↦ [x,0,y]
↦ [2x,0,2y,1]
ring S=(0,T),(a,b,c,d),ws(1,2,3,4);
minpoly = T^2+1;
ideal i=a2+b,c2+T^2*d2; i=std(i);
qring Q=i;
show(Q);
↦ // qring: (0,T),(a,b,c,d),(ws(1,2,3,4),C);
↦ // minpoly = (T2+1)
↦ // quotient ring from ideal:
↦ _[1]=a2+b
↦ _[2]=c2-d2
map F=r,a2,b^2,3*c3;
show(F);
↦ // i-th variable of preimage ring is mapped to @map[i]
↦ // @map                 [1]  map from r
↦ @map[1]=a2
↦ @map[2]=b2
↦ @map[3]=3*c3
// Apply 'show' to i (which does not belong to the basering) by typing
// ring r; ideal i=xy,x3-y2; ring Q; show(r,"i");
```

D.2.3.6 showrecursive

Procedure from library `inout.lib` (see Section D.2.3 [inout_lib], page 304).

Usage: showrecursive(id,p[ord]); id= any object of basering, p= product of variables and ord=string (any allowed ordstr)

Display: display 'id' in a recursive format as a polynomial in the variables occurring in p with coefficients in the remaining variables. This is done by mapping to a ring with parameters [and ordering 'ord', if a 3rd argument is present (default: ord="dp")] and applying procedure 'show'

Return: no return value

Example:

```
LIB "inout.lib";
ring r=2,(a,b,c,d,x,y),ds;
poly f=y+ax2+bx3+cx2y2+dxy3;
showrecursive(f,x);
↦ // poly, 4 monomial(s)
↦ (b)*x3+(a+cy2)*x2+(dy3)*x+(y)
showrecursive(f,xy,"lp");
↦ // poly, 5 monomial(s)
↦ (b)*x3+(c)*x2y2+(a)*x2+(d)*xy3+y
```

D.2.3.7 split

Procedure from library `inout.lib` (see Section D.2.3 [inout_lib], page 304).

Usage: split(s[,n]); s string, n integer

Return: same string, split into lines of length n separated by \ (default: n=pagewidth)

Note: may be used in connection with lprint

Example:

```
LIB "inout.lib";
ring r= 0,(x,y,z),ds;
poly f = (x+y+z)^4;
split(string(f),50);
↦ x4+4x3y+6x2y2+4xy3+y4+4x3z+12x2yz+12xy2z+4y3z+6x\
↦ 2z2+12xyz2+6y2z2+4xz3+4yz3+z4
split(lprint(f));
↦   x4+4x3y+6x2y2+4xy3+y4+4x3z+12x2yz+12xy2z+4y3z+6x2z2+12xyz2+6y2z2+4xz3+4\
   yz3\
↦ +z4
```

D.2.3.8 tab

Procedure from library `inout.lib` (see Section D.2.3 [inout_lib], page 304).

Usage: tab(n); n integer

Return: string of n space tabs

Example:

```
LIB "inout.lib";
for(int n=0; n<=5; n=n+1)
{ tab(5-n)+"*"+tab(n)+"+"+tab(n)+"*"; }
↦      *+*
↦     * + *
↦    *  +  *
↦   *   +   *
↦  *    +    *
↦ *     +     *
```

D.2.3.9 writelist

Procedure from library `inout.lib` (see Section D.2.3 [inout_lib], page 304).

Usage: writelist(file,name,L); file,name strings (file-name, list-name), L a list.

Create: a file with name ‘file‘, write the content of the list L into it and call the list ‘name‘, keeping the list structure

Return: no return value

Note: The syntax of writelist uses and is similar to the syntax of the write command of Singular which does not manage lists properly. If (file,name) = ("listfile","L1"), writelist creates (resp. appends if listfile exists) a file with name listfile and stores there the list L under the name L1. The Singular command execute(read("listfile")); assigns the content of L (stored in listfile) to a list L1.
On a UNIX system, write(">file",...) overwrites an existing file ‘file‘ while write("file",...) and write(">>file",...) append.

Example:

```
LIB "inout.lib";
ring r;
ideal i=x,y,z;
list k="Hi",nameof(basering),i,37;
writelist("zumSpass","lustig",k);
read("zumSpass");
↦ list lustig;
↦    lustig[1]=
↦ Hi;
↦    lustig[2]=
↦ r;
↦    lustig[3]=
↦ x,y,z;
↦    lustig[4]=
↦ 37;
↦
list L=res(i,0);                    //resolution of the ideal i
writelist("res_list","res-name",L); "";
↦
read("res_list");
↦ list res-name;
↦    res-name[1]=
↦ z,y,x;
↦    res-name[2]=
↦ -y*gen(1)+z*gen(2),-x*gen(1)+z*gen(3),-x*gen(2)+y*gen(3);
↦    res-name[3]=
↦ x*gen(1)-y*gen(2)+z*gen(3);
↦
// execute(read("res_list")); would create a list with name res-name,
// which is the resolution of i (the same content as L)
system("sh","/bin/rm res_list zumSpass");
↦ 0
// Under UNIX, this removes the files 'res_list' and 'zumSpass'
// Type help system; to get more information about the shell escape
// If your operating system does not accept the shell escape, you
// must remove the just created files 'zumSpass' and 'res_list' directly
```

D.2.3.10 pause

Procedure from library `inout.lib` (see Section D.2.3 [inout_lib], page 304).

Usage: pause([prompt]) prompt string

Return: none

Purpose: interrupt the execution of commands until user input

Note: pause is useful in procedures in connection with printlevel to interrupt the computation and to display intermediate results.

Example:

```
LIB "inout.lib";
// can only be shown interactively, try the following commands:
// pause("press <return> to continue");
// pause();
```

```
// In the following pocedure TTT, xxx is printed and the execution of
// TTT is stopped until the return-key is pressed, if printlevel>0.
// xxx may be any result of a previous computation or a comment, etc:
//
// proc TTT
// { int pp = printlevel-voice+2;  //pp=0 if printlevel=0 and if TTT is
//    ....                         //not called from another procedure
//    if( pp>0 )
//    {
//       print( xxx );
//       pause("press <return> to continue");
//    }
//     ....
// }
```

See also: Section 5.3.6 [printlevel], page 221; Section 5.1.105 [read], page 190.

D.2.4 poly_lib

Library: poly.lib

Purpose: Procedures for Manipulating Polys, Ideals, Modules

Authors: O. Bachmann, G.-M: Greuel, A. Fruehbis

Procedures:

D.2.4.1 cyclic

Procedure from library `poly.lib` (see Section D.2.4 [poly_lib], page 311).

Usage: cyclic(n); n integer

Return: ideal of cyclic n-roots from 1-st n variables of basering

Example:

```
LIB "poly.lib";
ring r=0,(u,v,w,x,y,z),lp;
cyclic(nvars(basering));
↦ _[1]=u+v+w+x+y+z
↦ _[2]=uv+uz+vw+wx+xy+yz
↦ _[3]=uvw+uvz+uyz+vwx+wxy+xyz
↦ _[4]=uvwx+uvwz+uvyz+uxyz+vwxy+wxyz
↦ _[5]=uvwxy+uvwxz+uvwyz+uvxyz+uwxyz+vwxyz
↦ _[6]=uvwxyz-1
homog(cyclic(5),z);
↦ _[1]=u+v+w+x+y
↦ _[2]=uv+uy+vw+wx+xy
↦ _[3]=uvw+uvy+uxy+vwx+wxy
↦ _[4]=uvwx+uvwy+uvxy+uwxy+vwxy
↦ _[5]=uvwxy-z5
```

D.2.4.2 katsura

Procedure from library `poly.lib` (see Section D.2.4 [poly_lib], page 311).

Usage: katsura([n]): n integer

Return: katsura(n) : n-th katsura ideal of
(1) newly created and set ring (32003, x(0..n), dp), if nvars(basering) < n
(2) basering, if nvars(basering) >= n
katsura() : katsura ideal of basering

Example:

```
LIB "poly.lib";
ring r; basering;
↦ //   characteristic : 32003
↦ //   number of vars : 3
↦ //        block   1 : ordering dp
↦ //                  : names    x y z
↦ //        block   2 : ordering C
katsura();
↦ _[1]=x+2y+2z-1
↦ _[2]=x2+2y2+2z2-x
↦ _[3]=2xy+2yz-y
katsura(4); basering;
↦ _[1]=x(0)+2*x(1)+2*x(2)+2*x(3)-1
↦ _[2]=x(0)^2+2*x(1)^2+2*x(2)^2+2*x(3)^2-x(0)
↦ _[3]=2*x(0)*x(1)+2*x(1)*x(2)+2*x(2)*x(3)-x(1)
↦ _[4]=x(1)^2+2*x(0)*x(2)+2*x(1)*x(3)-x(2)
↦ //   characteristic : 32003
↦ //   number of vars : 5
↦ //        block   1 : ordering dp
↦ //                  : names    x(0) x(1) x(2) x(3) x(4)
↦ //        block   2 : ordering C
```

D.2.4.3 freerank

Procedure from library `poly.lib` (see Section D.2.4 [poly_lib], page 311).

Usage: freerank(M[,any]); M=poly/ideal/vector/module/matrix

Compute: rank of module presented by M in case it is free.
By definition this is vdim(coker(M)/m*coker(M)) if coker(M) is free, where m = maximal ideal of the variables of the basering and M is considered as matrix.
(the 0-module is free of rank 0)

Return: rank of coker(M) if coker(M) is free and -1 else;
in case of a second argument return a list:
L[1] = rank of coker(M) or -1
L[2] = minbase(M)

Note: freerank(syz(M)); computes the rank of M if M is free (and -1 else)

Example:

```
LIB "poly.lib";
ring r;
ideal i=x;
module M=[x,0,1],[-x,0,-1];
freerank(M);          // should be 2, coker(M) is not free
↦ 2
freerank(syz (M),"");
↦ [1]:
```

```
↦     1
↦  [2]:
↦     _[1]=gen(2)+gen(1)
// [1] should be 1, coker(syz(M))=M is free of rank 1
// [2] should be gen(2)+gen(1) (minimal relation of M)
freerank(i);
↦ -1
freerank(syz(i));     // should be 1, coker(syz(i))=i is free of rank 1
↦ 1
```

D.2.4.4 is_homog

Procedure from library `poly.lib` (see Section D.2.4 [poly_lib], page 311).

Usage: is_homog(id); id poly/ideal/vector/module/matrix

Return: integer which is 1 if input is homogeneous (resp. weighted homogeneous if the monomial ordering consists of one block of type ws,Ws,wp or Wp, assuming that all weights are positive) and 0 otherwise

Note: A vector is homogeneous, if the components are homogeneous of same degree, a module/matrix is homogeneous if all column vectors are homogeneous
//*** ergaenzen, wenn Matrizen-Spalten Gewichte haben

Example:

```
LIB "poly.lib";
ring r = 0,(x,y,z),wp(1,2,3);
is_homog(x5-yz+y3);
↦ 0
ideal i = x6+y3+z2, x9-z3;
is_homog(i);
↦ 1
ring s = 0,(a,b,c),ds;
vector v = [a2,0,ac+bc];
vector w = [a3,b3,c4];
is_homog(v);
↦ 1
is_homog(w);
↦ 0
```

D.2.4.5 is_zero

Procedure from library `poly.lib` (see Section D.2.4 [poly_lib], page 311).

Usage: is_zero(M[,any]); M=poly/ideal/vector/module/matrix

Return: integer, 1 if coker(M)=0 resp. 0 if coker(M)!=0, where M is considered as matrix.
If a second argument is given, return a list:
L[1] = 1 if coker(M)=0 resp. 0 if coker(M)!=0
L[2] = dim(M)

Example:

```
LIB "poly.lib";
ring r;
module m = [x],[y],[1,z];
is_zero(m,1);
```

```
↦ [1]:
↦    0
↦ [2]:
↦    2
qring q = std(ideal(x2+y3+z2));
ideal j = x2+y3+z2-37;
is_zero(j);
↦ 1
```

D.2.4.6 lcm

Procedure from library `poly.lib` (see Section D.2.4 [poly_lib], page 311).

Usage: lcm(p[,q]); p int/intvec q a list of integers or
p poly/ideal q a list of polynomials

Return: the least common multiple of the common entries of p and q:
- of type int if p is an int/intvec
- of type poly if p is a poly/ideal

Example:

```
LIB "poly.lib";
ring  r = 0,(x,y,z),lp;
poly  p = (x+y)*(y+z);
poly  q = (z4+2)*(y+z);
lcm(p,q);
↦ xyz4+2xy+xz5+2xz+y2z4+2y2+yz5+2yz
ideal i=p,q,y+z;
lcm(i,p);
↦ xyz4+2xy+xz5+2xz+y2z4+2y2+yz5+2yz
lcm(2,3,6);
↦ 6
lcm(2..6);
↦ 60
```

D.2.4.7 maxcoef

Procedure from library `poly.lib` (see Section D.2.4 [poly_lib], page 311).

Usage: maxcoef(f); f poly/ideal/vector/module/matrix

Return: maximal length of coefficient of f of type int (by counting the length of the string of each coefficient)

Example:

```
LIB "poly.lib";
ring r= 0,(x,y,z),ds;
poly g = 345x2-1234567890y+7/4z;
maxcoef(g);
↦ 10
ideal i = g,10/1234567890;
maxcoef(i);
↦ 11
// since i[2]=1/123456789
```

D.2.4.8 maxdeg

Procedure from library `poly.lib` (see Section D.2.4 [poly_lib], page 311).

Usage: maxdeg(id); id poly/ideal/vector/module/matrix

Return: int/intmat, each component equals maximal degree of monomials in the corresponding component of id, independent of ring ordering (maxdeg of each var is 1).
Of type int if id is of type poly, of type intmat else

Note: proc maxdeg1 returns 1 integer, the absolute maximum; moreover, it has an option for computing weighted degrees

Example:

```
LIB "poly.lib";
ring r = 0,(x,y,z),wp(-1,-2,-3);
poly f = x+y2+z3;
deg(f);             //deg; returns weighted degree (in case of 1 block)!
↦ -1
maxdeg(f);
↦ 3
matrix m[2][2]=f+x10,1,0,f^2;
maxdeg(m);
↦ 10,0,
↦ -1,6
```

D.2.4.9 maxdeg1

Procedure from library `poly.lib` (see Section D.2.4 [poly_lib], page 311).

Usage: maxdeg1(id[,v]); id=poly/ideal/vector/module/matrix, v=intvec

Return: integer, maximal [weighted] degree of monomials of id independent of ring ordering, maxdeg1 of i-th variable is v[i] (default: v=1..1).

Note: This proc returns one integer while maxdeg returns, in general, a matrix of integers. For one polynomial and if no intvec v is given maxdeg is faster

Example:

```
LIB "poly.lib";
ring r = 0,(x,y,z),wp(-1,-2,-3);
poly f = x+y2+z3;
deg(f);            //deg returns weighted degree (in case of 1 block)!
↦ -1
maxdeg1(f);
↦ 3
intvec v = ringweights(r);
maxdeg1(f,v);                        //weighted maximal degree
↦ -1
matrix m[2][2]=f+x10,1,0,f^2;
maxdeg1(m,v);                        //absolute weighted maximal degree
↦ 0
```

D.2.4.10 mindeg

Procedure from library `poly.lib` (see Section D.2.4 [poly_lib], page 311).

Usage: mindeg(id); id poly/ideal/vector/module/matrix

Return: minimal degree/s of monomials of id, independent of ring ordering (mindeg of each variable is 1) of type int if id of type poly, else of type intmat.

Note: proc mindeg1 returns one integer, the absolute minimum; moreover it has an option for computing weighted degrees.

Example:

```
LIB "poly.lib";
ring r = 0,(x,y,z),ls;
poly f = x5+y2+z3;
ord(f);                  // ord returns weighted order of leading term!
↦ 3
mindeg(f);               // computes minimal degree
↦ 2
matrix m[2][2]=x10,1,0,f^2;
mindeg(m);               // computes matrix of minimum degrees
↦ 10,0,
↦ -1,4
```

D.2.4.11 mindeg1

Procedure from library `poly.lib` (see Section D.2.4 [poly_lib], page 311).

Usage: mindeg1(id[,v]); id=poly/ideal/vector/module/matrix, v=intvec

Return: integer, minimal [weighted] degree of monomials of id independent of ring ordering, mindeg1 of i-th variable is v[i] (default v=1..1).

Note: This proc returns one integer while mindeg returns, in general, a matrix of integers. For one polynomial and if no intvec v is given mindeg is faster.

Example:

```
LIB "poly.lib";
ring r = 0,(x,y,z),ls;
poly f = x5+y2+z3;
ord(f);                  // ord returns weighted order of leading term!
↦ 3
intvec v = 1,-3,2;
mindeg1(f,v);            // computes minimal weighted degree
↦ -6
matrix m[2][2]=x10,1,0,f^2;
mindeg1(m,1..3);         // computes absolute minimum of weighted degrees
↦ -1
```

D.2.4.12 normalize

Procedure from library `poly.lib` (see Section D.2.4 [poly_lib], page 311).

Usage: normalize(id); id=poly/vector/ideal/module

Return: object of same type with leading coefficient equal to 1

Example:

```
LIB "poly.lib";
ring r = 0,(x,y,z),ls;
poly f = 2x5+3y2+4z3;
normalize(f);
↦ z3+3/4y2+1/2x5
module m=[9xy,0,3z3],[4z,6y,2x];
normalize(m);
↦ _[1]=z3*gen(3)+3xy*gen(1)
↦ _[2]=z*gen(1)+3/2y*gen(2)+1/2x*gen(3)
ring s = 0,(x,y,z),(c,ls);
module m=[9xy,0,3z3],[4z,6y,2x];
normalize(m);
↦ _[1]=[xy,0,1/3z3]
↦ _[2]=[z,3/2y,1/2x]
```

D.2.4.13 rad_con

Procedure from library poly.lib (see Section D.2.4 [poly_lib], page 311).

Usage: rad_con(g,I); g polynomial, I ideal

Return: 1 (TRUE) (type int) if g is contained in the radical of I
0 (FALSE) (type int) otherwise

Example:

```
LIB "poly.lib";
ring R=0,(x,y,z),dp;
ideal I=x2+y2,z2;
poly f=x4+y4;
rad_con(f,I);
↦ 0
ideal J=x2+y2,z2,x4+y4;
poly g=z;
rad_con(g,I);
↦ 1
```

D.2.4.14 content

Procedure from library poly.lib (see Section D.2.4 [poly_lib], page 311).

Usage: content(f); f polynomial/vector

Return: number, the content (greatest common factor of coefficients) of the polynomial/vector f

Example:

```
LIB "poly.lib";
ring r=0,(x,y,z),(c,lp);
content(3x2+18xy-27xyz);
↦ 3
vector v=[3x2+18xy-27xyz,15x2+12y4,3];
content(v);
↦ 3
```

D.2.4.15 numerator

Procedure from library `poly.lib` (see Section D.2.4 [poly_lib], page 311).

Usage: numerator(n); n number

Return: number, the numerator of n

Example:

```
LIB "poly.lib";
ring r = 0,x, dp;
number n = 3/2;
numerator(n);
↦ 3
```

See also: Section 5.1.7 [cleardenom], page 127; Section D.2.4.14 [content], page 317; Section D.2.4.16 [denominator], page 318.

D.2.4.16 denominator

Procedure from library `poly.lib` (see Section D.2.4 [poly_lib], page 311).

Usage: denominator(n); n number

Return: number, the denominator of n

Example:

```
LIB "poly.lib";
ring r = 0,x, dp;
number n = 3/2;
denominator(n);
↦ 2
```

See also: Section 5.1.7 [cleardenom], page 127; Section D.2.4.14 [content], page 317; Section D.2.4.16 [denominator], page 318.

D.2.4.17 mod2id

Procedure from library `poly.lib` (see Section D.2.4 [poly_lib], page 311).

Usage: mod2id(M,vpos); M matrix, vpos intvec

Assume: vpos is an integer vector such that gen(i) corresponds to var(vpos[i]).
The basering contains variables var(vpos[i]) which do not occur in M.

Return: ideal I in which each gen(i) from the module is replaced by var(vpos[i]) and all monomials var(vpos[i])*var(vpos[j]) have been added to the generating set of I.

Note: This procedure should be used in the following situation: one wants to pass to a ring with new variables, say e(1),..,e(s), which correspond to the components gen(1),..,gen(s) of the module M such that e(i)*e(j)=0 for all i,j.
The new ring should already exist and be the current ring

Example:

```
LIB "poly.lib";
ring r=0,(e(1),e(2),x,y,z),(dp(2),ds(3));
module mo=x*gen(1)+y*gen(2);
intvec iv=2,1;
mod2id(mo,iv);
```

```
↦ _[1]=e(2)^2
↦ _[2]=e(1)*e(2)
↦ _[3]=e(1)^2
↦ _[4]=e(1)*y+e(2)*x
```

D.2.4.18 id2mod

Procedure from library `poly.lib` (see Section D.2.4 [poly_lib], page 311).

Usage: id2mod(I,vpos); I ideal, vpos intvec

Return: module corresponding to the ideal by replacing var(vpos[i]) by gen(i) and omitting all generators var(vpos[i])*var(vpos[j])

Note: * This procedure only makes sense if the ideal contains all var(vpos[i])*var(vpos[j]) as monomial generators and all other generators of I are linear combinations of the var(vpos[i]) over the ring in the other variables.
* This is the inverse procedure to mod2id and should be applied only to ideals created by mod2id using the same intvec vpos (possibly after a standard basis computation)

Example:

```
LIB "poly.lib";
ring r=0,(e(1),e(2),x,y,z),(dp(2),ds(3));
ideal i=e(2)^2,e(1)*e(2),e(1)^2,e(1)*y+e(2)*x;
intvec iv=2,1;
id2mod(i,iv);
↦ _[1]=x*gen(1)+y*gen(2)
```

D.2.4.19 substitute

Procedure from library `poly.lib` (see Section D.2.4 [poly_lib], page 311).

Usage: - case 1: typeof(#[1])==poly:
substitute (I,v,f[,v1,f1,v2,f2,...]); I object of basering which can be mapped, v,v1,v2,.. ring variables, f,f1,f2,... poly
- case 2: typeof(#[1])==ideal: substitute1 (I,v,f); I object of basering which can be mapped, v ideal of ring variables, f ideal

Return: object of same type as I,
- case 1: ring variable v,v1,v2,... substituted by polynomials f,f1,f2,..., in this order
- case 2: ring variables in v substituted by polynomials in f: v[i] is substituted by f[i], i=1,...,i=min(size(v),ncols(f))

Note: this procedure extends the built-in command subst which substitutes ring variables only by monomials

Example:

```
LIB "poly.lib";
ring r = 0,(b,c,t),dp;
ideal I = -bc+4b2c2t,bc2t-5b2c;
substitute(I,c,b+c,t,0,b,b-1);
↦ _[1]=-b2-bc+2b+c-1
↦ _[2]=-5b3-5b2c+15b2+10bc-15b-5c+5
ideal v = c,t,b;
ideal f = b+c,0,b-1;
substitute(I,v,f);
```

```
↦ _[1]=-b2-bc+2b+c-1
↦ _[2]=-5b3-5b2c+15b2+10bc-15b-5c+5
```

D.2.4.20 subrInterred

Procedure from library `poly.lib` (see Section D.2.4 [poly_lib], page 311).

Usage: subrInterred(mon,sm,iv);
sm: ideal in a ring r with n + s variables,
e.g. x_1,..,x_n and t_1,..,t_s
mon: ideal with monomial generators (not divisible by any of the t_i) such that sm is contained in the module k[t_1,..,t_s]*mon[1]+..+k[t_1,..,t_s]*mon[size(mon)]
iv: intvec listing the variables which are supposed to be used as x_i

Return: list l:
l[1]=the monomials from mon in the order used
l[2]=their coefficients after interreduction
l[3]=l[1]*l[2]

Purpose: Do interred only w.r.t. a subset of variables.
The procedure returns an interreduced system of generators of sm considered as a k[t_1,..,t_s]-submodule of the free module k[t_1,..,t_s]*mon[1]+..+k[t_1,..,t_s]*mon[size(mon)]).

Example:

```
LIB "poly.lib";
ring r=0,(x,y,z),dp;
ideal i=x^2+x*y^2,x*y+x^2*y,z;
ideal j=x^2+x*y^2,x*y,z;
ideal mon=x^2,z,x*y;
intvec iv=1,3;
subrInterred(mon,i,iv);
↦ [1]:
↦    _[1,1]=z
↦    _[1,2]=xy
↦    _[1,3]=x2
↦ [2]:
↦    _[1]=gen(1)
↦    _[2]=y2*gen(2)-gen(2)
↦    _[3]=y*gen(2)+gen(3)
↦ [3]:
↦    _[1,1]=z
↦    _[1,2]=xy3-xy
↦    _[1,3]=xy2+x2
subrInterred(mon,j,iv);
↦ [1]:
↦    _[1,1]=z
↦    _[1,2]=xy
↦    _[1,3]=x2
↦ [2]:
↦    _[1]=gen(1)
↦    _[2]=gen(2)
↦    _[3]=gen(3)
↦ [3]:
```

```
↦     _[1,1]=z
↦     _[1,2]=xy
↦     _[1,3]=x2
```

D.2.4.21 hilbPoly

Procedure from library `poly.lib` (see Section D.2.4 [poly_lib], page 311).

Usage: hilbPoly(I) I a homogeneous ideal

Return: the Hilbert polynomial of basering/I as an intvec v=v_0,...,v_r such that the Hilbert polynomial is (v_0+v_1*t+...v_r*t^r)/r!

Example:

```
LIB "poly.lib";
ring r = 0,(b,c,t,h),dp;
ideal I=
bct-t2h+2th2+h3,
bt3-ct3-t4+b2th+c2th-2bt2h+2ct2h+2t3h-bch2-2bth2+2cth2+2th3,
b2c2+bt2h-ct2h-t3h+b2h2+2bch2+c2h2-2bth2+2cth2+t2h2-2bh3+2ch3+2th3+3h4,
c2t3+ct4-c3th-2c2t2h-2ct3h-t4h+bc2h2-2c2th2-bt2h2+4t3h2+2bth3-2cth3-t2h3
+bh4-6th4-2h5;
hilbPoly(I);
↦ -11,10
```

D.2.5 random_lib

Library: random.lib

Purpose: Creating Random and Sparse Matrices, Ideals, Polys

Procedures:

D.2.5.1 genericid

Procedure from library `random.lib` (see Section D.2.5 [random_lib], page 321).

Usage: genericid(id,[,p,b]); id ideal/module, k,p,b integers

Return: system of generators of id which are generic, sparse, triagonal linear combinations of given generators with coefficients in [1,b] and sparseness p percent, bigger p being sparser (default: p=75, b=30000)

Note: For performance reasons try small bound b in characteristic 0

Example:

```
LIB "random.lib";
ring r=0,(t,x,y,z),ds;
ideal i= x3+y4,z4+yx,t+x+y+z;
genericid(i,0,10);
↦ _[1]=3t+3x+3y+3z+2xy+x3+y4+2z4
↦ _[2]=4t+4x+4y+4z+xy+z4
↦ _[3]=t+x+y+z
module m=[x,0,0,0],[0,y2,0,0],[0,0,z3,0],[0,0,0,t4];
print(genericid(m));
↦ x,      0,      0, 0,
↦ 17904y2,y2,     0, 0,
↦ 0,      24170z3,z3,0,
↦ 0,      0,      0, t4
```

D.2.5.2 randomid

Procedure from library `random.lib` (see Section D.2.5 [random_lib], page 321).

Usage: randomid(id,[k,b]); id ideal/module, b,k integers

Return: ideal/module having k generators which are random linear combinations of generators of id with coefficients in the interval [-b,b] (default: b=30000, k=size(id))

Note: For performance reasons try small bound b in characteristic 0

Example:

```
LIB "random.lib";
ring r=0,(x,y,z),dp;
randomid(maxideal(2),2,9);
↦ _[1]=-5x2-9xy+6y2-8xz-8yz+4z2
↦ _[2]=-9xy+2y2+xz+yz-z2
module m=[x,0,1],[0,y2,0],[y,0,z3];
show(randomid(m));
↦ // module, 3 generator(s)
↦ [1369x-11685y,-4481y2,-11685z3+1369]
↦ [-642x-13756y,25342y2,-13756z3-642]
↦ [2536x-6355y,8285y2,-6355z3+2536]
```

D.2.5.3 randommat

Procedure from library `random.lib` (see Section D.2.5 [random_lib], page 321).

Usage: randommat(n,m[,id,b]); n,m,b integers, id ideal

Return: nxm matrix, entries are random linear combinations of elements of id and coefficients in [-b,b]
[default: (id,b) = (maxideal(1),30000)]

Note: For performance reasons try small bound b in char 0

Example:

```
LIB "random.lib";
ring r=0,(x,y,z),dp;
matrix A=randommat(3,3,maxideal(2),9);
print(A);
↦ 9x2-2xy-8y2-9xz+yz+4z2, 9x2-4xy+y2-5xz+6yz-z2,   8x2+xy-9y2+2yz-8z2,
↦ -x2+5xy-8y2-7xz+4yz-3z2,x2+xy-4y2-xz+5z2,        5x2-8xy+8y2+6xz+yz+7z2,
↦ 4x2-5xy-6y2-4yz-5z2,    -4x2-6xy-4y2-8xz+3yz+5z2,2x2+3xy+y2+4xz-3yz+2z2
A=randommat(2,3);
print(A);
↦ 15276x+9897y+7526z,  6495x-24178y+11295z,-5745x-14754y+15979z,
↦ 20788x-28366y-20283z,24911x-10978y+3341z,12412x+11216y+15344z
```

D.2.5.4 sparseid

Procedure from library `random.lib` (see Section D.2.5 [random_lib], page 321).

Usage: sparseid(k,u[,o,p,b]); k,u,o,p,b integers

Return: ideal having k generators, each of degree d, u<=d<=o, p percent of terms in degree d are 0, the remaining have random coefficients in the interval [1,b], (default: o=u=d, p=75, b=30000)

Example:

```
LIB "random.lib";
ring r = 0,(a,b,c,d),ds;
sparseid(2,3);"";
↦ _[1]=12773a3+24263a2c+20030abc+17904b2c+26359c3
↦ _[2]=24004a3+6204b2c+24170bc2+19505c2d+21962bd2
↦
sparseid(3,0,4,90,9);
↦ _[1]=1+4a2+8b2c+3c3+4a3b+4a2b2+5abc2+3ac3
↦ _[2]=a+a2+7ab2+6a2c+3c3+5a3b+9ab3+2c4+3c3d+8ad3
↦ _[3]=5a+ab+2ac2+2b3c+8abcd
```

D.2.5.5 sparsematrix

Procedure from library `random.lib` (see Section D.2.5 [random_lib], page 321).

Usage: sparsematrix(n,m,o[,u,pe,pp,b]); n,m,o,u,pe,pp,b integers

Return: nxm matrix, about pe percent of the entries are 0, the remaining are random polynomials of degree d, u<=d<=o, with pp percent of the terms being 0, the remaining have random coefficients in the interval [1,b] [default: (pe,u,pp,b) = (0,50,75,100)]

Example:

```
LIB "random.lib";
ring r = 0,(a,b,c,d),dp;
// sparse matrix of sparse polys of degree <=2:
print(sparsematrix(3,4,2));"";
↦ 14ab+20bc+79cd+30b,    32a2+97bc+5b,      0,             0,
↦ 0,                     0,                 6c2+16b+64c+76,0,
↦ 17a2+30ab+94bc+19b+45d,88a2+44bc+13d2+31a,59ac,          0
↦
// dense matrix of sparse linear forms:
print(sparsematrix(3,3,1,1,0,55,9));
↦ 9b+7c+8d,9b+9d,5a,
↦ 7c+d,    a+6b, 2b+2d,
↦ 9a+5b+9c,2a+9d,2d
```

D.2.5.6 sparsemat

Procedure from library `random.lib` (see Section D.2.5 [random_lib], page 321).

Usage: sparsemat(n,m[,p,b]); n,m,p,b integers

Return: nxm integer matrix, p percent of the entries are 0, the remaining are random coefficients >=1 and <= b; [defaults: (p,b) = (75,1)]

Example:

```
LIB "random.lib";
sparsemat(5,5);"";
↦ 0,0,0,0,0,
↦ 0,1,0,0,1,
↦ 0,0,0,1,0,
↦ 0,1,0,0,0,
↦ 0,1,0,1,1
↦
```

```
sparsemat(5,5,95);"";
↦ 1,0,0,0,0,
↦ 0,0,0,0,0,
↦ 0,0,0,0,0,
↦ 0,0,0,0,0,
↦ 0,0,0,1,0
↦
sparsemat(5,5,5);"";
↦ 1,1,1,1,1,
↦ 1,1,1,1,1,
↦ 1,1,1,1,1,
↦ 1,0,1,1,1,
↦ 1,1,1,1,0
↦
sparsemat(5,5,50,100);
↦ 0,17,24,80,0,
↦ 0,13,30,45,0,
↦ 19,0,0,0,0,
↦ 93,0,23,0,69,
↦ 0,88,44,31,0
```

D.2.5.7 sparsepoly

Procedure from library `random.lib` (see Section D.2.5 [random_lib], page 321).

Usage: sparsepoly(u[,o,p,b]); u,o,p,b integers

Return: poly having only terms in degree d, u<=d<=o, p percent of the terms in degree d are 0, the remaining have random coefficients in [1,b), (defaults: o=u=d, p=75, b=30000)

Example:

```
LIB "random.lib";
ring r=0,(x,y,z),dp;
sparsepoly(5);"";
↦ 24263xy4+24170x4z+21962x3yz+26642xy3z+5664xy2z2+17904xz4
↦
sparsepoly(3,5,90,9);
↦ 8x3z2+2y3z2+3xyz3+2xy3+yz3+xy2
```

D.2.5.8 sparsetriag

Procedure from library `random.lib` (see Section D.2.5 [random_lib], page 321).

Usage: sparsetriag(n,m[,p,b]); n,m,p,b integers

Return: nxm lower triagonal integer matrix, diagonal entries equal to 1, about p percent of lower diagonal entries are 0, the remaining are random integers >=1 and <= b; [defaults: (p,b) = (75,1)]

Example:

```
LIB "random.lib";
sparsetriag(5,7);"";
↦ 1,0,0,0,0,0,0,
↦ 0,1,0,0,0,0,0,
↦ 0,1,1,0,0,0,0,
```

```
↦ 0,0,0,1,0,0,0,
↦ 1,1,0,0,1,0,0
↦
sparsetriag(7,5,90);"";
↦ 1,0,0,0,0,
↦ 0,1,0,0,0,
↦ 0,1,1,0,0,
↦ 0,0,0,1,0,
↦ 0,0,0,0,1,
↦ 0,0,0,1,0,
↦ 0,1,0,0,0
↦
sparsetriag(5,5,0);"";
↦ 1,0,0,0,0,
↦ 1,1,0,0,0,
↦ 1,1,1,0,0,
↦ 1,1,1,1,0,
↦ 1,1,1,1,1
↦
sparsetriag(5,5,50,100);
↦ 1,0,0,0,0,
↦ 73,1,0,0,0,
↦ 0,79,1,0,0,
↦ 14,0,0,1,0,
↦ 0,48,23,0,1
```

D.2.5.9 triagmatrix

Procedure from library `random.lib` (see Section D.2.5 [random_lib], page 321).

Usage: triagmatrix(n,m,o[,u,pe,pp,b]); n,m,o,u,pe,pp,b integers

Return: nxm lower triagonal matrix, diagonal entries equal to 1, about p percent of lower diagonal entries are 0, the remaining are random polynomials of degree d, u<=d<=o, with pp percent of the terms being 0, the remaining have random coefficients in the interval [1,b] [default: (pe,u,pp,b) = (0,50,75,100)]

Example:

```
LIB "random.lib";
ring r = 0,(a,b,c,d),dp;
// sparse triagonal matrix of sparse polys of degree <=2:
print(triagmatrix(3,4,2));"";
↦ 1,                                  0,0,0,
↦ 52ac+54cd+14c,                      1,0,0,
↦ 17a2+19b2+45ac+94bc+50b+87c+54d+21,0,1,0
↦
// dense triagonal matrix of sparse linear forms:
print(triagmatrix(3,3,1,1,0,55,9));
↦ 1,       0,    0,
↦ 7a+8d,   1,    0,
↦ 9b+7c+4d,7b+9d,1
```

D.2.5.10 randomLast

Procedure from library `random.lib` (see Section D.2.5 [random_lib], page 321).

Usage: randomLast(b); b int

Return: ideal = maxideal(1), but the last variable is exchanged by a random linear combination of all variables, with coefficients in the interval [-b,b].

Example:

```
LIB "random.lib";
ring  r = 0,(x,y,z),lp;
ideal i = randomLast(10);
i;
↦ i[1]=x
↦ i[2]=y
↦ i[3]=-x+z
```

D.2.5.11 randomBinomial

Procedure from library `random.lib` (see Section D.2.5 [random_lib], page 321).

Usage: randomBinomial(k,u[,o,b]); k,u,o,b integers

Return: binomial ideal, k homogeneous generators of degree d, u<=d<=o, with randomly chosen monomials and coefficients in the interval [-b,b] (default: u=o, b=10).

Example:

```
LIB "random.lib";
ring  r = 0,(x,y,z),lp;
ideal i = randomBinomial(4,5,6);
i;
↦ i[1]=-x4z-xz4
↦ i[2]=8x2y3+8xy3z
↦ i[3]=-4x2y2z2-4xy5
↦ i[4]=5x3yz2+5xz5
```

D.2.6 ring_lib

Library: ring.lib

Purpose: Manipulating Rings and Maps

Procedures:

D.2.6.1 changechar

Procedure from library `ring.lib` (see Section D.2.6 [ring_lib], page 326).

Usage: changechar(newr,c[,r]); newr,c=strings, r=ring

Create: create a new ring with name ‘newr‘ and make it the basering if r is an existing ring [default: r=basering].
The new ring differs from the old ring only in the characteristic. If, say, (newr,c) = ("R","0,A") and the ring r exists, the new basering will have name R, characteristic 0 and one parameter A.

Return: No return value

Note: Works for qrings if map from old_char to new_char is implemented This proc uses ’execute’ or calls a procedure using ’execute’. If you use it in your own proc, let the local names of your proc start with `@`.

Example:

```
LIB "ring.lib";
ring r=0,(x,y,u,v),(dp(2),ds);
changechar("R","2,A"); R;"";
↦ // basering is now R
↦ //   characteristic : 2
↦ //   1 parameter    : A
↦ //   minpoly        : 0
↦ //   number of vars : 4
↦ //        block   1 : ordering dp
↦ //                  : names    x y
↦ //        block   2 : ordering ds
↦ //                  : names    u v
↦ //        block   3 : ordering C
↦
changechar("R1","32003",R); R1;
↦ // basering is now R1
↦ //   characteristic : 32003
↦ //   number of vars : 4
↦ //        block   1 : ordering dp
↦ //                  : names    x y
↦ //        block   2 : ordering ds
↦ //                  : names    u v
↦ //        block   3 : ordering C
kill R,R1;
if(system("with","Namespaces")) {
if( nameof(Current) == "Ring" ) {
kill Top::R,Top::R1;
} else {
kill Ring::R,Ring::R1;
}
}
```

D.2.6.2 changeord

Procedure from library `ring.lib` (see Section D.2.6 [ring_lib], page 326).

Usage: changeord(newr,o[,r]); newr,o=strings, r=ring/qring

Create: create a new ring with name ‘newr‘ and make it the basering if r is an existing ring/qring [default: r=basering].
The new ring differs from the old ring only in the ordering. If, say, (newr,o) = ("R","wp(2,3),dp") and the ring r exists and has >=3 variables, the new basering will have name R and ordering wp(2,3),dp.

Return: No return value

Note: This proc uses ’execute’ or calls a procedure using ’execute’. If you use it in your own proc, let the local names of your proc start with `@`.

Example:

```
LIB "ring.lib";
ring r=0,(x,y,u,v),(dp(2),ds);
changeord("R","wp(2,3),dp"); R; "";
↦ // basering is now R
```

```
↦ //   characteristic : 0
↦ //   number of vars : 4
↦ //        block   1 : ordering wp
↦ //                  : names    x y
↦ //                  : weights  2 3
↦ //        block   2 : ordering dp
↦ //                  : names    u v
↦ //        block   3 : ordering C
↦
ideal i = x^2,y^2-u^3,v;
qring Q = std(i);
changeord("Q'","lp",Q); Q';
↦ // basering is now Q'
↦ //   characteristic : 0
↦ //   number of vars : 4
↦ //        block   1 : ordering lp
↦ //                  : names    x y u v
↦ //        block   2 : ordering C
↦ // quotient ring from ideal
↦ _[1]=v
↦ _[2]=x2
↦ _[3]=y2-u3
kill R,Q,Q';
if(system("with","Namespaces")) {
if( nameof(Current) == "Ring" ) {
kill Top::R,Top::Q';
} else {
kill Ring::R,Ring::Q';
}
}
```

D.2.6.3 changevar

Procedure from library `ring.lib` (see Section D.2.6 [ring_lib], page 326).

Usage: changevar(newr,vars[,r]); newr,vars=strings, r=ring/qring

Create: creates a new ring with name 'newr' and makes it the basering if r is an existing ring/qring [default: r=basering].
The new ring differs from the old ring only in the variables. If, say, (newr,vars) = ("R","t()") and the ring r exists and has n variables, the new basering will have name R and variables t(1),...,t(n).
If vars = "a,b,c,d", the new ring will have the variables a,b,c,d.

Return: No return value

Note: This procedure is useful in connection with the procedure ringtensor, when a conflict between variable names must be avoided. This proc uses 'execute' or calls a procedure using 'execute'. If you use it in your own proc, let the local names of your proc start with `@`.

Example:

```
LIB "ring.lib";
ring r=0,(x,y,u,v),(dp(2),ds);
ideal i = x^2,y^2-u^3,v;
```

```
qring Q = std(i);
setring(r);
changevar("R","A()"); R; "";
↦ // basering is now R
↦ //   characteristic : 0
↦ //   number of vars : 4
↦ //        block   1 : ordering dp
↦ //                  : names    A(1) A(2)
↦ //        block   2 : ordering ds
↦ //                  : names    A(3) A(4)
↦ //        block   3 : ordering C
↦
changevar("Q'","a,b,c,d",Q); Q';
↦ // basering is now Q'
↦ //   characteristic : 0
↦ //   number of vars : 4
↦ //        block   1 : ordering dp
↦ //                  : names    a b
↦ //        block   2 : ordering ds
↦ //                  : names    c d
↦ //        block   3 : ordering C
↦ // quotient ring from ideal
↦ _[1]=d
↦ _[2]=a2
↦ _[3]=b2-c3
kill R,Q,Q';
if(system("with","Namespaces")) {
if( nameof(Current) == "Ring" ) {
kill Top::R,Top::Q';
} else {
kill Ring::R,Ring::Q';
}
}
```

D.2.6.4 defring

Procedure from library `ring.lib` (see Section D.2.6 [ring_lib], page 326).

Usage: defring(s1,s2,n,s3,s4); s1..s4=strings, n=integer

Create: Define a ring with name 's1', characteristic 's2', ordering 's4' and n variables with names derived from s3 and make it the basering. If s3 is a single letter, say s3="a", and if n<=26 then a and the following n-1 letters from the alphabet (cyclic order) are taken as variables. If n>26 or if s3 is a single letter followed by (, say s3="T(", the variables are T(1),...,T(n).

Return: No return value

Note: This proc is useful for defining a ring in a procedure. This proc uses 'execute' or calls a procedure using 'execute'. If you use it in your own proc, let the local names of your proc start with `@`.

Example:

```
LIB "ring.lib";
defring("r","0",5,"u","ls"); r; "";
```

```
↦ // basering is now: r
↦ //   characteristic : 0
↦ //   number of vars : 5
↦ //        block   1 : ordering ls
↦ //                  : names    u v w x y
↦ //        block   2 : ordering C
↦
defring("R","2,A",10,"x(","dp(3),ws(1,2,3),ds"); R;
↦ // basering is now: R
↦ //   characteristic : 2
↦ //   1 parameter    : A
↦ //   minpoly        : 0
↦ //   number of vars : 10
↦ //        block   1 : ordering dp
↦ //                  : names    x(1) x(2) x(3)
↦ //        block   2 : ordering ws
↦ //                  : names    x(4) x(5) x(6)
↦ //                  : weights     1    2    3
↦ //        block   3 : ordering ds
↦ //                  : names    x(7) x(8) x(9) x(10)
↦ //        block   4 : ordering C
kill r,R;
if(system("with","Namespaces")) {
if( nameof(Current) == "Ring" ) {
kill Top::r,Top::R;
} else {
kill Ring::r,Ring::R;
}
}
```

D.2.6.5 defrings

Procedure from library `ring.lib` (see Section D.2.6 [ring_lib], page 326).

Usage: defrings(n,[p]); n,p integers

Create: Defines a ring with name Sn, characteristic p, ordering ds and n variables x,y,z,a,b,...if n<=26 (resp. x(1..n) if n>26) and makes it the basering (default: p=32003)

Return: No return value

Example:

```
LIB "ring.lib";
defrings(5,0); S5; "";
↦ // basering is now: ring S5=0,(x,y,z,a,b),ds;
↦ //   characteristic : 0
↦ //   number of vars : 5
↦ //        block   1 : ordering ds
↦ //                  : names    x y z a b
↦ //        block   2 : ordering C
↦
defrings(30); S30;
↦ // basering is now: ring S30=32003,x(1..30),ds;
↦ //   characteristic : 32003
↦ //   number of vars : 30
```

```
↦ //        block   1 : ordering ds
↦ //                  : names    x(1) x(2) x(3) x(4) x(5) x(6) x(7) x(8) x(\
   9) x(10) x(11) x(12) x(13) x(14) x(15) x(16) x(17) x(18) x(19) x(20) x(21\
   ) x(22) x(23) x(24) x(25) x(26) x(27) x(28) x(29) x(30)
↦ //        block   2 : ordering C
kill S5, S30;
if(system("with","Namespaces")) {
if( nameof(Current) == "Ring" ) {
kill Top::S5,Top::S30;
} else {
kill Ring::S5,Ring::S30;
}
}
```

D.2.6.6 defringp

Procedure from library `ring.lib` (see Section D.2.6 [ring_lib], page 326).

Usage: defringp(n,[p]); n,p=integers

Create: defines a ring with name Pn, characteristic p, ordering dp and n variables x,y,z,a,b,...if n<=26 (resp. x(1..n) if n>26) and makes it the basering (default: p=32003)

Return: No return value

Example:

```
LIB "ring.lib";
defringp(5,0); P5; "";
↦ // basering is now: ring P5=0,(x,y,z,a,b),dp;
↦ //   characteristic : 0
↦ //   number of vars : 5
↦ //        block   1 : ordering dp
↦ //                  : names    x y z a b
↦ //        block   2 : ordering C
↦
defringp(30); P30;
↦ // basering is now: ring P30=32003,x(1..30),dp;
↦ //   characteristic : 32003
↦ //   number of vars : 30
↦ //        block   1 : ordering dp
↦ //                  : names    x(1) x(2) x(3) x(4) x(5) x(6) x(7) x(8) x(\
   9) x(10) x(11) x(12) x(13) x(14) x(15) x(16) x(17) x(18) x(19) x(20) x(21\
   ) x(22) x(23) x(24) x(25) x(26) x(27) x(28) x(29) x(30)
↦ //        block   2 : ordering C
kill P5, P30;
if(system("with","Namespaces")) {
if( nameof(Current) == "Ring" ) {
kill Top::P5,Top::P30;
} else {
kill Ring::P5,Ring::P30;
}
}
```

D.2.6.7 extendring

Procedure from library `ring.lib` (see Section D.2.6 [ring_lib], page 326).

Usage: extendring(na,n,va,o[iv,i,r]); na,va,o=strings,
n,i=integers, r=ring, iv=intvec of positive integers or iv=0

Create: Define a ring with name 'na' which extends the ring r by adding n new variables in front of [after, if i!=0] the old variables and make it the basering [default: (i,r)=(0,basering)].
– The characteristic is the characteristic of r.
– The new vars are derived from va. If va is a single letter, say va="T", and if n<=26 then T and the following n-1 letters from T..Z..T (resp. T(1..n) if n>26) are taken as additional variables. If va is a single letter followed by (, say va="x(", the new variables are x(1),...,x(n).
– The ordering is the product ordering between the ordering of r and an ordering derived from 'o' [and iv].
- If o contains a 'c' or a 'C' in front resp. at the end this is taken for the whole ordering in front resp. at the end. If o does not contain a 'c' or a 'C' the same rule applies to ordstr(r).
- If no intvec iv is given, or if iv=0, o may be any allowed ordstr, like "ds" or "dp(2),wp(1,2,3),Ds(2)" or "ds(a),dp(b),ls" if a and b are globally (!) defined integers and if a+b+1<=n. If, however, a and b are local to a proc calling extendring, the intvec iv must be used to let extendring know the values of a and b
- If an intvec iv !=0 is given, iv[1],iv[2],... is taken for the 1st, 2nd,... block of o, if o contains no substring "w" or "W" i.e. no weighted ordering (in the above case o="ds,dp,ls" and iv=a,b).
If o contains a weighted ordering (only one (!) weighted block is allowed) iv[1] is taken as size for the weight-vector, the next iv[1] values of iv are taken as weights and the remaining values of iv as block-size for the remaining non-weighted blocks. e.g. o="dp,ws,Dp,ds", iv=3,2,3,4,2,5 creates the ordering dp(2),ws(2,3,4),Dp(5),ds

Return: No return value

Note: This proc is useful for adding deformation parameters.
This proc uses 'execute' or calls a procedure using 'execute'. If you use it in your own proc, let the local names of your proc start with @ (see the file HelpForProc)

Example:

```
LIB "ring.lib";
ring r=0,(x,y,z),ds;
show(r);"";
↦ // ring: (0),(x,y,z),(ds(3),C);
↦ // minpoly = 0
↦ // objects belonging to this ring:
↦
//blocksize is derived from no of vars:
int t=5;
extendring("R1",t,"a","dp");         //t global: "dp" -> "dp(5)"
↦ // basering is now R1
show(R1); "";
↦ // ring: (0),(a,b,c,d,e,x,y,z),(dp(5),ds(3),C);
↦ // minpoly = 0
↦ // objects belonging to this ring:
↦
extendring("R2",4,"T(","c,dp",1,r);    //"dp" -> "c,..,dp(4)"
↦ // basering is now R2
show(R2);"";
↦ // ring: (0),(x,y,z,T(1),T(2),T(3),T(4)),(c,ds(3),dp(4));
```

```
↦ // minpoly = 0
↦ // objects belonging to this ring:
↦
//no intvec given, blocksize given: given blocksize is used:
extendring("R3",4,"T(","dp(2)",0,r);   // "dp(2)" -> "dp(2)"
↦ // basering is now R3
show(R3);"";
↦ // ring: (0),(T(1),T(2),T(3),T(4),x,y,z),(dp(2),ds(5),C);
↦ // minpoly = 0
↦ // objects belonging to this ring:
↦
//intvec given: weights and blocksize is derived from given intvec
//(no specification of a blocksize in the given ordstr is allowed!)
//if intvec does not cover all given blocks, the last block is used
//for the remaining variables, if intvec has too many components,
//the last ones are ignored
intvec v=3,2,3,4,1,3;
extendring("R4",10,"A","ds,ws,Dp,dp",v,0,r);
↦ // basering is now R4
//v covers 3 blocks: v[1] (=3) : no of components of ws
//next v[1] values (=v[2..4]) give weights
//remaining components of v are used for the remaining blocks
show(R4);
↦ // ring: (0),(A,B,C,D,E,F,G,H,I,J,x,y,z),(ds(1),ws(2,3,4),Dp(3),dp(3),ds(\
   3),C);
↦ // minpoly = 0
↦ // objects belonging to this ring:
kill r,R1,R2,R3,R4;
if(system("with","Namespaces")) {
if( nameof(Current) == "Ring" ) {
kill Top::R1,Top::R2,Top::R3,Top::R4;
} else {
kill Ring::R1,Ring::R2,Ring::R3,Ring::R4;
}
}
```

D.2.6.8 fetchall

Procedure from library `ring.lib` (see Section D.2.6 [ring_lib], page 326).

Usage: fetchall(R[,s]); R=ring/qring, s=string

Create: fetch all objects of ring R (of type poly/ideal/vector/module/number/ matrix) into the basering.
If no 3rd argument is present, the names are the same as in R. If, say, f is a poly in R and the 3rd argument is the string "R", then f is mapped to f_R etc.

Return: no return value

Note: As fetch, this procedure maps the 1st, 2nd, ... variable of R to the 1st, 2nd, ... variable of the basering.
The 3rd argument is useful in order to avoid conflicts of names, the empty string is allowed

Caution: fetchall does not work inside a procedure.
It does not work if R contains a map.

Example:

```
LIB "ring.lib";
// The example is not shown since fetchall does not work in a procedure;
// (and hence not in the example procedure). Try the following commands:
//   ring R=0,(x,y,z),dp;
//   ideal j=x,y2,z2;
//   matrix M[2][3]=1,2,3,x,y,z;
//   j; print(M);
//   ring S=0,(a,b,c),ds;
//   fetchall(R);       //map from R to S: x->a, y->b, z->c;
//   names(S);
//   j; print(M);
//   fetchall(S,"1");   //identity map of S: copy objects, change names
//   names(S);
//   kill R,S;
```

D.2.6.9 imapall

Procedure from library `ring.lib` (see Section D.2.6 [ring_lib], page 326).

Usage: imapall(R[,s]); R=ring/qring, s=string

Create: map all objects of ring R (of type poly/ideal/vector/module/number/ matrix) into the basering, by applying imap to all objects of R. If no 3rd argument is present, the names are the same as in R. If, say, f is a poly in R and the 3rd argument is the string "R", then f is mapped to f_R etc.

Return: no return value

Note: As imap, this procedure maps the variables of R to the variables with the same name in the basering, the other variables are mapped to 0. The 3rd argument is useful in order to avoid conflicts of names, the empty string is allowed

Caution: imapall does not work inside a procedure
It does not work if R contains a map

Example:

```
LIB "ring.lib";
// The example is not shown since imapall does not work in a procedure
// (and hence not in the example procedure). Try the following commands:
//   ring R=0,(x,y,z,u),dp;
//   ideal j=x,y,z,u2+ux+z;
//   matrix M[2][3]=1,2,3,x,y,uz;
//   j; print(M);
//   ring S=0,(a,b,c,x,z,y),ds;
//   imapall(R);         //map from R to S: x->x, y->y, z->z, u->0
//   names(S);
//   j; print(M);
//   imapall(S,"1");     //identity map of S: copy objects, change names
//   names(S);
//   kill R,S;
```

D.2.6.10 mapall

Procedure from library `ring.lib` (see Section D.2.6 [ring_lib], page 326).

Usage: mapall(R,i[,s]); R=ring/qring, i=ideal of basering, s=string

Create: map all objects of ring R (of type poly/ideal/vector/module/number/ matrix, map) into the basering, by mapping the j-th variable of R to the j-th generator of the ideal i. If no 3rd argument is present, the names are the same as in R. If, say, f is a poly in R and the 3rd argument is the string "R", then f is mapped to f_R etc.

Return: no return value.

Note: This procedure has the same effect as defining a map, say psi, by map psi=R,i; and then applying psi to all objects of R. In particular, maps from R to some ring S are composed with psi, creating thus a map from the basering to S.
mapall may be combined with copyring to change vars for all objects. The 3rd argument is useful in order to avoid conflicts of names, the empty string is allowed.

Caution: mapall does not work inside a procedure.

Example:

```
LIB "ring.lib";
// The example is not shown since mapall does not work in a procedure
// (and hence not in the example procedure). Try the following commands:
//   ring R=0,(x,y,z),dp;
//   ideal j=x,y,z;
//   matrix M[2][3]=1,2,3,x,y,z;
//   map phi=R,x2,y2,z2;
//   ring S=0,(a,b,c),ds;
//   ideal i=c,a,b;
//   mapall(R,i);         //map from R to S: x->c, y->a, z->b
//   names(S);
//   j; print(M); phi;    //phi maps R to S: x->c2, y->a2, z->b2
//   ideal i1=a2,a+b,1;
//   mapall(R,i1,"");   //map from R to S: x->a2, y->a+b, z->1
//   names(S);
//   j_; print(M_); phi_;
//   changevar("T","x()",R);  //change vars in R and call result T
//   mapall(R,maxideal(1));       //identity map from R to T
//   names(T);
//   j; print(M); phi;
//   kill R,S,T;
```

D.2.6.11 ord_test

Procedure from library `ring.lib` (see Section D.2.6 [ring_lib], page 326).

Usage: ord_test(r); r ring

Return: int 1 (resp. -1, resp. 0) if ordering of r is global (resp. local, resp. mixed)

Example:

```
LIB "ring.lib";
ring R = 0,(x,y),dp;
ring S = 0,(u,v),ls;
ord_test(R);
↦ 1
ord_test(S);
↦ -1
```

```
ord_test(R+S);
↦ 0
```

D.2.6.12 ringtensor

Procedure from library `ring.lib` (see Section D.2.6 [ring_lib], page 326).

Usage: ringtensor(s,r1,r2,...); s=string, r1,r2,...=rings

Create: A new base ring with name 's' if r1,r2,... are existing rings. If, say, s = "R" and the rings r1,r2,... exist, the new ring will have name R, variables from all rings r1,r2,... and as monomial ordering the block (product) ordering of r1,r2,... . Hence, R is the tensor product of the rings r1,r2,... with ordering matrix equal to the direct sum of the ordering matrices of r1,r2,...

Return: no return value

Note: The characteristic of the new ring will be that of r1. The names of variables in the rings r1,r2,... should differ (if a name, say x, occurs in r1 and r2, then, in the new ring r, x always refers to the variable with name x in r1, there is no access to x in r2). The procedure works also for quotient rings ri, if the characteristic of ri is compatible with the characteristic of r1 (i.e. if imap from ri to r1 is implemented)
This proc uses 'execute' or calls a procedure using 'execute'. If you use it in your own proc, let the local names of your proc start with `@` (see the file HelpForProc)

Example:

```
LIB "ring.lib";
ring r=32003,(x,y,u,v),dp;
ring s=0,(a,b,c),wp(1,2,3);
ring t=0,x(1..5),(c,ls);
ringtensor("R",r,s,t);
↦ // basering is now R
type R;
↦ // R                    [0]  *ring
↦ //   characteristic : 32003
↦ //   number of vars : 12
↦ //        block   1 : ordering dp
↦ //                  : names    x y u v
↦ //        block   2 : ordering wp
↦ //                  : names    a b c
↦ //                  : weights  1 2 3
↦ //        block   3 : ordering ls
↦ //                  : names    x(1) x(2) x(3) x(4) x(5)
↦ //        block   4 : ordering C
setring s;
ideal i = a2+b3+c5;
changevar("S","x,y,z");       //change vars of sand make S the basering
↦ // basering is now S
qring qS =std(fetch(s,i));    //create qring of S mod i (mapped to S)
changevar("T","d,e,f,g,h",t); //change vars of t and make T the basering
↦ // basering is now T
qring qT=std(d2+e2-f3);       //create qring of T mod d2+e2-f3
ringtensor("Q",s,qS,t,qT);
↦ // basering is now Q
type Q;
```

```
↦ // Q                    [0]  *qring
↦ //   characteristic : 0
↦ //   number of vars : 16
↦ //        block   1 : ordering wp
↦ //                  : names    a b c
↦ //                  : weights  1 2 3
↦ //        block   2 : ordering wp
↦ //                  : names    x y z
↦ //                  : weights  1 2 3
↦ //        block   3 : ordering ls
↦ //                  : names    x(1) x(2) x(3) x(4) x(5)
↦ //        block   4 : ordering ls
↦ //                  : names    d e f g h
↦ //        block   5 : ordering C
↦ // quotient ring from ideal
↦ _[1]=f3-e2-d2
↦ _[2]=z5+y3+x2
kill R,Q,S,T;
if(system("with","Namespaces")) {
if( nameof(Current) == "Ring" ) {
kill Top::R,Top::Q,Top::S,Top::T;
} else {
kill Ring::R,Ring::Q,Ring::S,Ring::T;
}
}
```

D.2.6.13 ringweights

Procedure from library `ring.lib` (see Section D.2.6 [ring_lib], page 326).

Usage: ringweights(P); P=name of an existing ring (true name, not a string)

Return: intvec consisting of the weights of the variables of P, as they appear when typing P;.

Note: This is useful when enlarging P but keeping the weights of the old variables.

Example:

```
LIB "ring.lib";
ring r0 = 0,(x,y,z),dp;
ringweights(r0);
↦ 1,1,1
ring r1 = 0,x(1..5),(ds(3),wp(2,3));
ringweights(r1);"";
↦ 1,1,1,2,3
↦
// an example for enlarging the ring, keeping the first weights:
intvec v = ringweights(r1),6,2,3,4,5;
ring R = 0,x(1..10),(a(v),dp);
ordstr(R);
↦ a(1,1,1,2,3,6,2,3,4,5),dp(10),C
```

D.3 Linear algebra

D.3.1 matrix_lib

Library: matrix.lib

Purpose: Elementary Matrix Operations

Procedures:

D.3.1.1 compress

Procedure from library `matrix.lib` (see Section D.3.1 [matrix_lib], page 338).

Usage: compress(A); A matrix/ideal/module/intmat/intvec

Return: same type, zero columns/generators from A deleted
(if A=intvec, zero elements are deleted)

Example:

```
LIB "matrix.lib";
ring r=0,(x,y,z),ds;
matrix A[3][4]=1,0,3,0,x,0,z,0,x2,0,z2,0;
print(A);
↦ 1, 0,3, 0,
↦ x, 0,z, 0,
↦ x2,0,z2,0
print(compress(A));
↦ 1, 3,
↦ x, z,
↦ x2,z2
module m=module(A); show(m);
↦ // module, 4 generator(s)
↦ [1,x,x2]
↦ [0]
↦ [3,z,z2]
↦ [0]
show(compress(m));
↦ // module, 2 generator(s)
↦ [1,x,x2]
↦ [3,z,z2]
intmat B[3][4]=1,0,3,0,4,0,5,0,6,0,7,0;
compress(B);
↦ 1,3,
↦ 4,5,
↦ 6,7
intvec C=0,0,1,2,0,3;
compress(C);
↦ 1,2,3
```

D.3.1.2 concat

Procedure from library `matrix.lib` (see Section D.3.1 [matrix_lib], page 338).

Usage: concat(A1,A2,..); A1,A2,... matrices

Return: matrix, concatenation of A1,A2,.... Number of rows of result matrix is max(nrows(A1),nrows(A2),...)

Example:

```
LIB "matrix.lib";
ring r=0,(x,y,z),ds;
matrix A[3][3]=1,2,3,x,y,z,x2,y2,z2;
matrix B[2][2]=1,0,2,0; matrix C[1][4]=4,5,x,y;
print(A);
↦ 1, 2, 3,
↦ x, y, z,
↦ x2,y2,z2
print(B);
↦ 1,0,
↦ 2,0
print(C);
↦ 4,5,x,y
print(concat(A,B,C));
↦ 1, 2, 3, 1,0,4,5,x,y,
↦ x, y, z, 2,0,0,0,0,0,
↦ x2,y2,z2,0,0,0,0,0,0
```

D.3.1.3 diag

Procedure from library `matrix.lib` (see Section D.3.1 [matrix_lib], page 338).

Usage: diag(p,n); p poly, n integer
diag(A); A matrix

Return: diag(p,n): diagonal matrix, p times unit matrix of size n.
diag(A) : n*m x n*m diagonal matrix with entries all the entries of the nxm matrix A, taken from the 1st row, 2nd row etc of A

Example:

```
LIB "matrix.lib";
ring r = 0,(x,y,z),ds;
print(diag(xy,4));
↦ xy,0, 0, 0,
↦ 0, xy,0, 0,
↦ 0, 0, xy,0,
↦ 0, 0, 0, xy
matrix A[3][2] = 1,2,3,4,5,6;
print(A);
↦ 1,2,
↦ 3,4,
↦ 5,6
print(diag(A));
↦ 1,0,0,0,0,0,
↦ 0,2,0,0,0,0,
↦ 0,0,3,0,0,0,
↦ 0,0,0,4,0,0,
↦ 0,0,0,0,5,0,
↦ 0,0,0,0,0,6
```

D.3.1.4 dsum

Procedure from library `matrix.lib` (see Section D.3.1 [matrix_lib], page 338).

Usage: dsum(A1,A2,..); A1,A2,... matrices

Return: matrix, direct sum of A1,A2,...

Example:

```
LIB "matrix.lib";
ring r = 0,(x,y,z),ds;
matrix A[3][3] = 1,2,3,4,5,6,7,8,9;
matrix B[2][2] = 1,x,y,z;
print(A);
↦ 1,2,3,
↦ 4,5,6,
↦ 7,8,9
print(B);
↦ 1,x,
↦ y,z
print(dsum(A,B));
↦ 1,2,3,0,0,
↦ 4,5,6,0,0,
↦ 7,8,9,0,0,
↦ 0,0,0,1,x,
↦ 0,0,0,y,z
```

D.3.1.5 flatten

Procedure from library `matrix.lib` (see Section D.3.1 [matrix_lib], page 338).

Usage: flatten(A); A matrix

Return: ideal, generated by all entries from A

Example:

```
LIB "matrix.lib";
ring r = 0,(x,y,z),ds;
matrix A[2][3] = 1,2,x,y,z,7;
print(A);
↦ 1,2,x,
↦ y,z,7
flatten(A);
↦ _[1]=1
↦ _[2]=2
↦ _[3]=x
↦ _[4]=y
↦ _[5]=z
↦ _[6]=7
```

D.3.1.6 genericmat

Procedure from library `matrix.lib` (see Section D.3.1 [matrix_lib], page 338).

Usage: genericmat(n,m[,id]); n,m=integers, id=ideal

Return: nxm matrix, with entries from id.

Note: if id has less than nxm elements, the matrix is filled with 0's, (default: id=maxideal(1)).
genericmat(n,m); creates the generic nxm matrix

Example:

```
LIB "matrix.lib";
ring R = 0,x(1..16),lp;
print(genericmat(3,3));      // the generic 3x3 matrix
↦ x(1),x(2),x(3),
↦ x(4),x(5),x(6),
↦ x(7),x(8),x(9)
ring R1 = 0,(a,b,c,d),dp;
matrix A = genericmat(3,4,maxideal(1)^3);
print(A);
↦ a3, a2b,a2c,a2d,
↦ ab2,abc,abd,ac2,
↦ acd,ad2,b3, b2c
int n,m = 3,2;
ideal i = ideal(randommat(1,n*m,maxideal(1),9));
print(genericmat(n,m,i));    // matrix of generic linear forms
↦ 4a-8b-2c-3d,-a+b-4c+5d,
↦ -8a-9b+c+7d,a-9b+9c+4d,
↦ 6a-5b+9c,   2a+8c+d
```

D.3.1.7 is_complex

Procedure from library `matrix.lib` (see Section D.3.1 [matrix_lib], page 338).

Usage: is_complex(c); c = list of size-compatible modules or matrices

Return: 1 if c[i]*c[i+1]=0 for all i, 0 if not, hence checking whether the list of matrices forms a complex.

Note: Ideals are treated internally as 1-line matrices.
If printlevel > 0, the position where c is not a complex is shown.

Example:

```
LIB "matrix.lib";
ring r  = 32003,(x,y,z),ds;
ideal i = x4+y5+z6,xyz,yx2+xz2+zy7;
list L  = nres(i,0);
is_complex(L);
↦ 1
L[4]    = matrix(i);
is_complex(L);
↦ 0
```

D.3.1.8 outer

Procedure from library `matrix.lib` (see Section D.3.1 [matrix_lib], page 338).

Usage: outer(A,B); A,B matrices

Return: matrix, outer (tensor) product of A and B

Example:

```
LIB "matrix.lib";
ring r=32003,(x,y,z),ds;
matrix A[3][3]=1,2,3,4,5,6,7,8,9;
matrix B[2][2]=x,y,0,z;
```

```
print(A);
↦ 1,2,3,
↦ 4,5,6,
↦ 7,8,9
print(B);
↦ x,y,
↦ 0,z
print(outer(A,B));
↦ x, y, 2x,2y,3x,3y,
↦ 0, z, 0, 2z,0, 3z,
↦ 4x,4y,5x,5y,6x,6y,
↦ 0, 4z,0, 5z,0, 6z,
↦ 7x,7y,8x,8y,9x,9y,
↦ 0, 7z,0, 8z,0, 9z
```

D.3.1.9 power

Procedure from library `matrix.lib` (see Section D.3.1 [matrix_lib], page 338).

Usage: power(A,n); A a square-matrix of type intmat or matrix, n=integer

Return: intmat resp. matrix, the n-th power of A

Note: for A=intmat and big n the result may be wrong because of int overflow

Example:

```
LIB "matrix.lib";
intmat A[3][3]=1,2,3,4,5,6,7,8,9;
print(power(A,3));"";
↦    468   576   684
↦   1062  1305  1548
↦   1656  2034  2412
↦
ring r=0,(x,y,z),dp;
matrix B[3][3]=0,x,y,z,0,0,y,z,0;
print(power(B,3));"";
↦ yz2,    xy2+x2z,y3+xyz,
↦ y2z+xz2,yz2,    0,
↦ y3+xyz, y2z+xz2,yz2
↦
```

D.3.1.10 skewmat

Procedure from library `matrix.lib` (see Section D.3.1 [matrix_lib], page 338).

Usage: skewmat(n[,id]); n integer, id ideal

Return: skew-symmetric nxn matrix, with entries from id
(default: id=maxideal(1))
skewmat(n); creates the generic skew-symmetric matrix

Note: if id has less than n*(n-1)/2 elements, the matrix is
filled with 0's,

Example:

```
LIB "matrix.lib";
ring R=0,x(1..5),lp;
print(skewmat(4));    // the generic skew-symmetric matrix
↦ 0,    x(1), x(2),x(3),
↦ -x(1),0,    x(4),x(5),
↦ -x(2),-x(4),0,   0,
↦ -x(3),-x(5),0,   0
ring R1 = 0,(a,b,c),dp;
matrix A=skewmat(4,maxideal(1)^2);
print(A);
↦ 0,  a2, ab, ac,
↦ -a2,0,  b2, bc,
↦ -ab,-b2,0,  c2,
↦ -ac,-bc,-c2,0
int n=3;
ideal i = ideal(randommat(1,n*(n-1) div 2,maxideal(1),9));
print(skewmat(n,i));  // skew matrix of generic linear forms
↦ 0,       4a+b-8c, -a+6b+c,
↦ -4a-b+8c,0,       -8a+2b-9c,
↦ a-6b-c,  8a-2b+9c,0
kill R1;
```

D.3.1.11 submat

Procedure from library `matrix.lib` (see Section D.3.1 [matrix_lib], page 338).

Usage: submat(A,r,c); A=matrix, r,c=intvec

Return: matrix, submatrix of A with rows specified by intvec r and columns specified by intvec c.

Example:

```
LIB "matrix.lib";
ring R=32003,(x,y,z),lp;
matrix A[4][4]=x,y,z,0,1,2,3,4,5,6,7,8,9,x2,y2,z2;
print(A);
↦ x,y, z, 0,
↦ 1,2, 3, 4,
↦ 5,6, 7, 8,
↦ 9,x2,y2,z2
intvec v=1,3,4;
matrix B=submat(A,v,1..3);
print(B);
↦ x,y, z,
↦ 5,6, 7,
↦ 9,x2,y2
```

D.3.1.12 symmat

Procedure from library `matrix.lib` (see Section D.3.1 [matrix_lib], page 338).

Usage: symmat(n[,id]); n integer, id ideal

Return: symmetric nxn matrix, with entries from id (default: id=maxideal(1))

Note: if id has less than n*(n+1)/2 elements, the matrix is filled with 0's, symmat(n); creates the generic symmetric matrix

Example:

```
LIB "matrix.lib";
ring R=0,x(1..10),lp;
print(symmat(4));    // the generic symmetric matrix
↦ 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)
ring R1 = 0,(a,b,c),dp;
matrix A=symmat(4,maxideal(1)^3);
print(A);
↦ a3, a2b,a2c,ab2,
↦ a2b,abc,ac2,b3,
↦ a2c,ac2,b2c,bc2,
↦ ab2,b3, bc2,c3
int n=3;
ideal i = ideal(randommat(1,n*(n+1) div 2,maxideal(1),9));
print(symmat(n,i));  // symmetric matrix of generic linear forms
↦ 4a-8b-2c,-a+b-4c, -8a-9b+c,
↦ -a+b-4c, a-9b+9c, 6a-5b+9c,
↦ -8a-9b+c,6a-5b+9c,2a+8c
kill R1;
```

D.3.1.13 tensor

Procedure from library `matrix.lib` (see Section D.3.1 [matrix_lib], page 338).

Usage: tensor(A,B); A,B matrices

Return: matrix, tensor product of A and B

Example:

```
LIB "matrix.lib";
ring r=32003,(x,y,z),(c,ds);
matrix A[3][3]=1,2,3,4,5,6,7,8,9;
matrix B[2][2]=x,y,0,z;
print(A);
↦ 1,2,3,
↦ 4,5,6,
↦ 7,8,9
print(B);
↦ x,y,
↦ 0,z
print(tensor(A,B));
↦ x, y, 2x,2y,3x,3y,
↦ 0, z, 0, 2z,0, 3z,
↦ 4x,4y,5x,5y,6x,6y,
↦ 0, 4z,0, 5z,0, 6z,
↦ 7x,7y,8x,8y,9x,9y,
↦ 0, 7z,0, 8z,0, 9z
```

D.3.1.14 unitmat

Procedure from library `matrix.lib` (see Section D.3.1 [matrix_lib], page 338).

Usage: unitmat(n); n integer >= 0

Return: nxn unit matrix

Note: needs a basering, diagonal entries are numbers (=1) in the basering

Example:

```
LIB "matrix.lib";
ring r=32003,(x,y,z),lp;
print(xyz*unitmat(4));
↦ xyz,0,  0,  0,
↦ 0,  xyz,0,  0,
↦ 0,  0,  xyz,0,
↦ 0,  0,  0,  xyz
print(unitmat(5));
↦ 1,0,0,0,0,
↦ 0,1,0,0,0,
↦ 0,0,1,0,0,
↦ 0,0,0,1,0,
↦ 0,0,0,0,1
```

D.3.1.15 gauss_col

Procedure from library `matrix.lib` (see Section D.3.1 [matrix_lib], page 338).

Usage: gauss_col(A[,e]); A a matrix, e any type

Return: - a matrix B, if called with one argument; B is the complete column- reduced upper-triangular normal form of A if A is constant, (resp. as far as this is possible if A is a polynomial matrix; no division by polynomials).
- a list L of two matrices, if called with two arguments; L satisfies L[1] = A * L[2] with L[1] the column-reduced form of A and L[2] the transformation matrix.

Note: * The procedure just applies interred to A with ordering (C,dp). The transformation matrix is obtained by applying 'lift'. This should be faster than the procedure colred.
* It should only be used with exact coefficient field (there is no pivoting and rounding error treatment).
* Parameters are allowed. Hence, if the entries of A are parameters, B is the column-reduced form of A over the rational function field.

Example:

```
LIB "matrix.lib";
ring r=(0,a,b),(A,B,C),dp;
matrix m[8][6]=
0,    2*C, 0,    0,  0,   0,
0,    -4*C,a*A,  0,  0,   0,
b*B,  -A,  0,    0,  0,   0,
-A,   B,   0,    0,  0,   0,
-4*C, 0,   B,    2,  0,   0,
2*A,  B,   0,    0,  0,   0,
0,    3*B, 0,    0,  2b,  0,
0,    AB,  0,    2*A,A,   2a;"";
```

```
↦
list L=gauss_col(m,1);
print(L[1]);
↦ 0,0,2*C, 0,       0,0,
↦ A,0,-4*C,0,       0,0,
↦ 0,0,-A,  (1/2b)*B,0,0,
↦ 0,0,B,   -1/2*A,  0,0,
↦ 0,1,0,   0,       0,0,
↦ 0,0,B,   A,       0,0,
↦ 0,0,0,   0,       1,0,
↦ 0,0,0,   0,       0,1
print(L[2]);
↦ 0,         0,        0,                1/2,      0,         0,
↦ 0,         0,        1,                0,        0,         0,
↦ 1/(a),     0,        0,                0,        0,         0,
↦ -1/(2a)*B, 1/2,      0,                C,        0,         0,
↦ 0,         0,        -3/(2b)*B,        0,        1/(2b),    0,
↦ 1/(2a2)*AB,-1/(2a)*A,(-2b+3)/(4ab)*AB,-1/(a)*AC,-1/(4ab)*A,1/(2a)
ring S=0,x,(c,dp);
matrix A[5][4] =
3, 1, 1, 1,
13, 8, 6,-7,
14,10, 6,-7,
7, 4, 3,-3,
2, 1, 0, 3;
print(gauss_col(A));
↦ 8/9,-5/9,-1/3,7/9,
↦ 1,  0,   0,   0,
↦ 0,  1,   0,   0,
↦ 0,  0,   1,   0,
↦ 0,  0,   0,   1
```

See also: Section D.3.1.24 [colred], page 351.

D.3.1.16 gauss_row

Procedure from library `matrix.lib` (see Section D.3.1 [matrix_lib], page 338).

Usage: gauss_row(A [,e]); A matrix, e any type

Return: - a matrix B, if called with one argument; B is the complete row- reduced lower-triangular normal form of A if A is constant, (resp. as far as this is possible if A is a polynomial matrix; no division by polynomials).
- a list L of two matrices, if called with two arguments; L satisfies L[1] = L[2] * A with L[1] the row-reduced form of A and L[2] the transformation matrix.

Note: * This procedure just applies gauss_col to the transposed matrix. The transformation matrix is obtained by applying lift. This should be faster than the procedure rowred.
* It should only be used with exact coefficient field (there is no pivoting and rounding error treatment).
* Parameters are allowed. Hence, if the entries of A are parameters, B is the row-reduced form of A over the rational function field.

Example:

```
LIB "matrix.lib";
```

```
ring r=(0,a,b),(A,B,C),dp;
matrix m[6][8]=
0, 0,  b*B, -A,-4C,2A,0, 0,
2C,-4C,-A,B, 0,  B, 3B,AB,
0,a*A,  0, 0, B,  0, 0, 0,
0, 0,  0, 0, 2,  0, 0, 2A,
0, 0,  0, 0, 0,  0, 2b, A,
0, 0,  0, 0, 0,  0, 0, 2a;"";
↦
print(gauss_row(m));"";
↦ 0,  A,   0,       0,     0,0,0,0,
↦ 0,  0,   0,       0,     1,0,0,0,
↦ 2*C,-4*C,-A,      B,     0,B,0,0,
↦ 0,  0,   (1/2b)*B,-1/2*A,0,A,0,0,
↦ 0,  0,   0,       0,     0,0,1,0,
↦ 0,  0,   0,       0,     0,0,0,1
↦
ring S=0,x,dp;
matrix A[4][5] =  3, 1,1,-1,2,
13, 8,6,-7,1,
14,10,6,-7,1,
7, 4,3,-3,3;
list L=gauss_row(A,1);
print(L[1]);
↦ 1/2,-7/3,-19/6,5/6,
↦ 1,  0,   0,    0,
↦ 0,  1,   0,    0,
↦ 0,  0,   1,    0,
↦ 0,  0,   0,    1
print(L[2]);
↦ 0,   -6,  -5,  1,
↦ -1/2,2/3, -1/6,-1/6,
↦ 1/2, -5/3,-5/6,1/6,
↦ 0,   13/3,11/3,-1/3
```

See also: Section D.3.1.23 [rowred], page 349.

D.3.1.17 addcol

Procedure from library `matrix.lib` (see Section D.3.1 [matrix_lib], page 338).

Usage: addcol(A,c1,p,c2); A matrix, p poly, c1, c2 positive integers

Return: matrix, A being modified by adding p times column c1 to column c2

Example:

```
LIB "matrix.lib";
ring r=32003,(x,y,z),lp;
matrix A[3][3]=1,2,3,4,5,6,7,8,9;
print(A);
↦ 1,2,3,
↦ 4,5,6,
↦ 7,8,9
print(addcol(A,1,xy,2));
↦ 1,xy+2, 3,
```

```
↦ 4,4xy+5,6,
↦ 7,7xy+8,9
```

D.3.1.18 addrow

Procedure from library `matrix.lib` (see Section D.3.1 [matrix_lib], page 338).

Usage: addcol(A,r1,p,r2); A matrix, p poly, r1, r2 positive integers

Return: matrix, A being modified by adding p times row r1 to row r2

Example:

```
LIB "matrix.lib";
ring r=32003,(x,y,z),lp;
matrix A[3][3]=1,2,3,4,5,6,7,8,9;
print(A);
↦ 1,2,3,
↦ 4,5,6,
↦ 7,8,9
print(addrow(A,1,xy,3));
↦ 1,   2,    3,
↦ 4,   5,    6,
↦ xy+7,2xy+8,3xy+9
```

D.3.1.19 multcol

Procedure from library `matrix.lib` (see Section D.3.1 [matrix_lib], page 338).

Usage: addcol(A,c,p); A matrix, p poly, c positive integer

Return: matrix, A being modified by multiplying column c with p

Example:

```
LIB "matrix.lib";
ring r=32003,(x,y,z),lp;
matrix A[3][3]=1,2,3,4,5,6,7,8,9;
print(A);
↦ 1,2,3,
↦ 4,5,6,
↦ 7,8,9
print(multcol(A,2,xy));
↦ 1,2xy,3,
↦ 4,5xy,6,
↦ 7,8xy,9
```

D.3.1.20 multrow

Procedure from library `matrix.lib` (see Section D.3.1 [matrix_lib], page 338).

Usage: multrow(A,r,p); A matrix, p poly, r positive integer

Return: matrix, A being modified by multiplying row r with p

Example:

```
LIB "matrix.lib";
ring r=32003,(x,y,z),lp;
matrix A[3][3]=1,2,3,4,5,6,7,8,9;
```

```
print(A);
↦ 1,2,3,
↦ 4,5,6,
↦ 7,8,9
print(multrow(A,2,xy));
↦ 1,  2,  3,
↦ 4xy,5xy,6xy,
↦ 7,  8,  9
```

D.3.1.21 permcol

Procedure from library `matrix.lib` (see Section D.3.1 [matrix_lib], page 338).

Usage: permcol(A,c1,c2); A matrix, c1,c2 positive integers

Return: matrix, A being modified by permuting column c1 and c2

Example:

```
LIB "matrix.lib";
ring r=32003,(x,y,z),lp;
matrix A[3][3]=1,x,3,4,y,6,7,z,9;
print(A);
↦ 1,x,3,
↦ 4,y,6,
↦ 7,z,9
print(permcol(A,2,3));
↦ 1,3,x,
↦ 4,6,y,
↦ 7,9,z
```

D.3.1.22 permrow

Procedure from library `matrix.lib` (see Section D.3.1 [matrix_lib], page 338).

Usage: permrow(A,r1,r2); A matrix, r1,r2 positive integers

Return: matrix, A being modified by permuting row r1 and r2

Example:

```
LIB "matrix.lib";
ring r=32003,(x,y,z),lp;
matrix A[3][3]=1,2,3,x,y,z,7,8,9;
print(A);
↦ 1,2,3,
↦ x,y,z,
↦ 7,8,9
print(permrow(A,2,1));
↦ x,y,z,
↦ 1,2,3,
↦ 7,8,9
```

D.3.1.23 rowred

Procedure from library `matrix.lib` (see Section D.3.1 [matrix_lib], page 338).

Usage: rowred(A[,e]); A matrix, e any type

Return: - a matrix B, being the row reduced form of A, if rowred is called with one argument. (as far as this is possible over the polynomial ring; no division by polynomials)
- a list L of two matrices, such that L[1] = L[2] * A with L[1] the row-reduced form of A and L[2] the transformation matrix (if rowred is called with two arguments).

Note: * This procedure is designed for teaching purposes mainly.
* The straight forward Gaussian algorithm is implemented in the library (no standard basis computation).
The transformation matrix is obtained by concatenating a unit matrix to A. proc gauss_row should be faster.
* It should only be used with exact coefficient field (there is no pivoting) over the polynomial ring (ordering lp or dp).
* Parameters are allowed. Hence, if the entries of A are parameters the computation takes place over the field of rational functions.

Example:

```
LIB "matrix.lib";
ring r=(0,a,b),(A,B,C),dp;
matrix m[6][8]=
0, 0,  b*B, -A,-4C,2A,0, 0,
2C,-4C,-A,B, 0,  B, 3B,AB,
0,a*A,  0, 0, B,  0, 0, 0,
0, 0,  0, 0, 2,  0, 0, 2A,
0, 0,  0, 0, 0,  0, 2b, A,
0, 0,  0, 0, 0,  0, 0, 2a;"";
↦
print(rowred(m));"";
↦ 0,  0,    0,    0, 1,0,  0,0,
↦ 0,  0,    0,    0, 0,0,  1,0,
↦ 0,  0,    0,    0, 0,0,  0,1,
↦ 0,  0,    (b)*B,-A,0,2*A,0,0,
↦ 2*C,-4*C, -A,   B, 0,B,  0,0,
↦ 0,  (a)*A,0,    0, 0,0,  0,0
↦
list L=rowred(m,1);
print(L[1]);
↦ 0,  0,    0,    0, 1,0,  0,0,
↦ 0,  0,    0,    0, 0,0,  1,0,
↦ 0,  0,    0,    0, 0,0,  0,1,
↦ 0,  0,    (b)*B,-A,0,2*A,0,0,
↦ 2*C,-4*C, -A,   B, 0,B,  0,0,
↦ 0,  (a)*A,0,    0, 0,0,  0,0
print(L[2]);
↦ 0,0,0,1/2,   0,        -1/(2a)*A,
↦ 0,0,0,0,     1/(2b),   -1/(4ab)*A,
↦ 0,0,0,0,     0,        1/(2a),
↦ 1,0,0,2*C,   0,        -2/(a)*AC,
↦ 0,1,0,0,     -3/(2b)*B,(-2b+3)/(4ab)*AB,
↦ 0,0,1,-1/2*B,0,        1/(2a)*AB
```

See also: Section D.3.1.16 [gauss_row], page 346.

D.3.1.24 colred

Procedure from library `matrix.lib` (see Section D.3.1 [matrix_lib], page 338).

Usage: colred(A[,e]); A matrix, e any type

Return: - a matrix B, being the column reduced form of A, if colred is called with one argument.
(as far as this is possible over the polynomial ring; no division by polynomials)
- a list L of two matrices, such that L[1] = A * L[2] with L[1] the column-reduced form of A and L[2] the transformation matrix (if colred is called with two arguments).

Note: * This procedure is designed for teaching purposes mainly.
* It applies rowred to the transposed matrix. proc gauss_col should be faster.
* It should only be used with exact coefficient field (there is no pivoting) over the polynomial ring (ordering lp or dp).
* Parameters are allowed. Hence, if the entries of A are parameters the computation takes place over the field of rational functions.

Example:

```
LIB "matrix.lib";
ring r=(0,a,b),(A,B,C),dp;
matrix m[8][6]=
0,    2*C, 0,    0,  0,   0,
0,    -4*C,a*A,  0,  0,   0,
b*B,  -A,  0,    0,  0,   0,
-A,   B,   0,    0,  0,   0,
-4*C, 0,   B,    2,  0,   0,
2*A,  B,   0,    0,  0,   0,
0,    3*B, 0,    0,  2b,  0,
0,    AB,  0,    2*A,A,   2a;"";
↦
print(colred(m));"";
↦ 0,0,0,0,    2*C, 0,
↦ 0,0,0,0,    -4*C,(a)*A,
↦ 0,0,0,(b)*B,-A,  0,
↦ 0,0,0,-A,   B,   0,
↦ 1,0,0,0,    0,   0,
↦ 0,0,0,2*A,  B,   0,
↦ 0,1,0,0,    0,   0,
↦ 0,0,1,0,    0,   0
↦
list L=colred(m,1);
print(L[1]);
↦ 0,0,0,0,    2*C, 0,
↦ 0,0,0,0,    -4*C,(a)*A,
↦ 0,0,0,(b)*B,-A,  0,
↦ 0,0,0,-A,   B,   0,
↦ 1,0,0,0,    0,   0,
↦ 0,0,0,2*A,  B,   0,
↦ 0,1,0,0,    0,   0,
↦ 0,0,1,0,    0,   0
print(L[2]);
↦ 0,       0,        0,     1,       0,               0,
↦ 0,       0,        0,     0,       1,               0,
↦ 0,       0,        0,     0,       0,               1,
```

```
↦ 1/2,      0,         0,     2*C,      0,                -1/2*B,
↦ 0,        1/(2b),    0,     0,        -3/(2b)*B,        0,
↦ -1/(2a)*A,-1/(4ab)*A,1/(2a),-2/(a)*AC,(-2b+3)/(4ab)*AB,1/(2a)*AB
```

See also: Section D.3.1.15 [gauss_col], page 345.

D.3.1.25 rm_unitrow

Procedure from library `matrix.lib` (see Section D.3.1 [matrix_lib], page 338).

Usage: rm_unitrow(A); A matrix (being col-reduced)

Return: matrix, obtained from A by deleting unit rows (having just one 1 and else 0 as entries) and associated columns

Example:

```
LIB "matrix.lib";
ring r=0,(A,B,C),dp;
matrix m[8][6]=
0,0,  0,   0, 2C, 0,
0,0,  0,   0, -4C,A,
A,-C2,0,   B, -A, 0,
0,0,  1/2B,-A,B,  0,
1,0,  0,   0, 0,  0,
0,0,  0,   2A,B,  0,
0,1,  0,   0, 0,  0,
0,0,  1,   0, 0,  0;
print(rm_unitrow(m));
↦ 0, 2C, 0,
↦ 0, -4C,A,
↦ B, -A, 0,
↦ -A,B,  0,
↦ 2A,B,  0
```

D.3.1.26 rm_unitcol

Procedure from library `matrix.lib` (see Section D.3.1 [matrix_lib], page 338).

Usage: rm_unitcol(A); A matrix (being row-reduced)

Return: matrix, obtained from A by deleting unit columns (having just one 1 and else 0 as entries) and associated rows

Example:

```
LIB "matrix.lib";
ring r=0,(A,B,C),dp;
matrix m[6][8]=
0,  0,    A,   0, 1,0,  0,0,
0,  0,  -C2,   0, 0,0,  1,0,
0,  0,    0,1/2B, 0,0,  0,1,
0,  0,    B,  -A, 0,2A, 0,0,
2C,-4C,  -A,   B, 0,B,  0,0,
0,  A,    0,   0, 0,0,  0,0;
print(rm_unitcol(m));
↦ 0, 0,  B, -A,2A,
↦ 2C,-4C,-A,B, B,
↦ 0, A,  0, 0, 0
```

D.3.2 linalg_lib

Library: linalg.lib

Purpose: Algorithmic Linear Algebra

Authors: Ivor Saynisch (ivs@math.tu-cottbus.de)
Mathias Schulze (mschulze@mathematik.uni-kl.de)

Procedures:

D.3.2.1 inverse

Procedure from library `linalg.lib` (see Section D.3.2 [linalg_lib], page 353).

Usage: inverse(A [,opt]); A a square matrix, opt integer

Return:

a matrix:
- the inverse matrix of A, if A is invertible;
- the 1x1 0-matrix if A is not invertible (in the polynomial ring!).
There are the following options:
- opt=0 or not given: heuristically best option from below
- opt=1 : apply std to (transpose(E,A)), ordering (C,dp).
- opt=2 : apply interred (transpose(E,A)), ordering (C,dp).
- opt=3 : apply lift(A,E), ordering (C,dp).

Note: parameters and minpoly are allowed; opt=2 is only correct for matrices with entries in a field

Example:

```
LIB "linalg.lib";
ring r=0,(x,y,z),lp;
matrix A[3][3]=
1,4,3,
1,5,7,
0,4,17;
print(inverse(A));"";
matrix B[3][3]=
y+1,  x+y,    y,
z,    z+1,    z,
y+z+2,x+y+z+2,y+z+1;
print(inverse(B));
print(B*inverse(B));
```

See also: Section D.3.2.2 [inverse_B], page 353; Section D.3.2.3 [inverse_L], page 354.

D.3.2.2 inverse_B

Procedure from library `linalg.lib` (see Section D.3.2 [linalg_lib], page 353).

Usage: inverse_B(A); A = square matrix

Return: list Inv with
- Inv[1] = matrix I and
- Inv[2] = poly p
such that I*A = unitmat(n)*p;

Note: p=1 if 1/det(A) is computable and p=det(A) if not;
the computation uses busadj.

Example:

```
LIB "linalg.lib";
ring r=0,(x,y),lp;
matrix A[3][3]=x,y,1,1,x2,y,x,6,0;
print(A);
list Inv=inverse_B(A);
print(Inv[1]);
print(Inv[2]);
print(Inv[1]*A);
```

See also: Section D.3.2.1 [inverse], page 353; Section D.3.2.3 [inverse_L], page 354.

D.3.2.3 inverse_L

Procedure from library `linalg.lib` (see Section D.3.2 [linalg_lib], page 353).

Usage: inverse_L(A); A = square matrix

Return: list Inv representing a left inverse of A, i.e
- Inv[1] = matrix I and
- Inv[2] = poly p
such that I*A = unitmat(n)*p;

Note: p=1 if 1/det(A) is computable and p=det(A) if not;
the computation computes first det(A) and then uses lift

Example:

```
LIB "linalg.lib";
ring r=0,(x,y),lp;
matrix A[3][3]=x,y,1,1,x2,y,x,6,0;
print(A);
list Inv=inverse_L(A);
print(Inv[1]);
print(Inv[2]);
print(Inv[1]*A);
```

See also: Section D.3.2.1 [inverse], page 353; Section D.3.2.2 [inverse_B], page 353.

D.3.2.4 sym_gauss

Procedure from library `linalg.lib` (see Section D.3.2 [linalg_lib], page 353).

Usage: sym_gauss(A); A = symmetric matrix

Return: matrix, diagonalisation with symmetric gauss algorithm

Example:

```
LIB "linalg.lib";
ring r=0,(x),lp;
matrix A[2][2]=1,4,4,15;
print(A);
print(sym_gauss(A));
```

D.3.2.5 orthogonalize

Procedure from library `linalg.lib` (see Section D.3.2 [linalg_lib], page 353).

Usage: orthogonalize(A); A = constant matrix

Return: matrix, orthogonal basis of the colum space of A

Example:

```
LIB "linalg.lib";
ring r=0,(x),lp;
matrix A[4][4]=5,6,12,4,7,3,2,6,12,1,1,2,6,4,2,10;
print(A);
print(orthogonalize(A));
```

D.3.2.6 diag_test

Procedure from library `linalg.lib` (see Section D.3.2 [linalg_lib], page 353).

Usage: diag_test(A); A = const square matrix

Return: int, 1 if A is diagonalisable, 0 if not
-1 no statement is possible, since A does not split.

Note: The test works only for split matrices, i.e if eigenvalues of A are in the ground field.
Does not work with parameters (uses factorize,gcd).

Example:

```
LIB "linalg.lib";
ring r=0,(x),dp;
matrix A[4][4]=6,0,0,0,0,0,6,0,0,6,0,0,0,0,0,6;
print(A);
diag_test(A);
```

D.3.2.7 busadj

Procedure from library `linalg.lib` (see Section D.3.2 [linalg_lib], page 353).

Usage: busadj(A); A = square matrix (of size nxn)

Return: list L:
L[1] contains the (n+1) coefficients of the characteristic
polynomial X of A, i.e.
X = L[1][1]+..+L[1][k]*t^(k-1)+..+(L[1][n+1])*t^n
L[2] contains the n (nxn)-matrices Hk which are the coefficients of
the busadjoint bA = adjoint(E*t-A) of A, i.e.
bA = (Hn-1)*t^(n-1)+...+Hk*t^k+...+H0, (Hk=L[2][k+1])

Example:

```
LIB "linalg.lib";
ring r = 0,(t,x),lp;
matrix A[2][2] = 1,x2,x,x2+3x;
print(A);
list L = busadj(A);
poly X = L[1][1]+L[1][2]*t+L[1][3]*t2; X;
matrix bA[2][2] = L[2][1]+L[2][2]*t;
print(bA);               //the busadjoint of A;
print(bA*(t*unitmat(2)-A));
```

D.3.2.8 charpoly

Procedure from library `linalg.lib` (see Section D.3.2 [linalg_lib], page 353).

Usage: charpoly(A[,v]); A square matrix, v string, name of a variable

Return: poly, the characteristic polynomial det(E*v-A)
(default: v=name of last variable)

Note: A must be independent of the variable v. The computation uses det. If printlevel>0, det(E*v-A) is diplayed recursively.

Example:

```
LIB "linalg.lib";
ring r=0,(x,t),dp;
matrix A[3][3]=1,x2,x,x2,6,4,x,4,1;
print(A);
charpoly(A,"t");
```

D.3.2.9 adjoint

Procedure from library `linalg.lib` (see Section D.3.2 [linalg_lib], page 353).

Usage: adjoint(A); A = square matrix

Return: adjoint matrix of A, i.e. Adj*A=det(A)*E

Note: computation uses busadj(A)

Example:

```
LIB "linalg.lib";
ring r=0,(t,x),lp;
matrix A[2][2]=1,x2,x,x2+3x;
print(A);
matrix Adj[2][2]=adjoint(A);
print(Adj);                    //Adj*A=det(A)*E
print(Adj*A);
```

D.3.2.10 det_B

Procedure from library `linalg.lib` (see Section D.3.2 [linalg_lib], page 353).

Usage: det_B(A); A any matrix

Return: returns the determinant of A

Note: the computation uses the busadj algorithm

Example:

```
LIB "linalg.lib";
ring r=0,(x),dp;
matrix A[10][10]=random(2,10,10)+unitmat(10)*x;
print(A);
det_B(A);
```

D.3.2.11 gaussred

Procedure from library `linalg.lib` (see Section D.3.2 [linalg_lib], page 353).

Usage: gaussred(A); A any constant matrix

Return: list Z: Z[1]=P , Z[2]=U , Z[3]=S , Z[4]=rank(A)
gives a row reduced matrix S, a permutation matrix P and a normalized lower triangular matrix U, with P*A=U*S

Note: This procedure is designed for teaching purposes mainly. The straight forward implementation in the interpreted library is not very efficient (no standard basis computation).

Example:

```
LIB "linalg.lib";
ring r=0,(x),dp;
matrix A[5][4]=1,3,-1,4,2,5,-1,3,1,3,-1,4,0,4,-3,1,-3,1,-5,-2;
print(A);
list Z=gaussred(A);   //construct P,U,S s.t. P*A=U*S
print(Z[1]);          //P
print(Z[2]);          //U
print(Z[3]);          //S
print(Z[4]);          //rank
print(Z[1]*A);        //P*A
print(Z[2]*Z[3]);     //U*S
```

D.3.2.12 gaussred_pivot

Procedure from library `linalg.lib` (see Section D.3.2 [linalg_lib], page 353).

Usage: gaussred_pivot(A); A any constant matrix

Return: list Z: Z[1]=P , Z[2]=U , Z[3]=S , Z[4]=rank(A)
gives n row reduced matrix S, a permutation matrix P and a normalized lower triangular matrix U, with P*A=U*S

Note: with row pivoting

Example:

```
LIB "linalg.lib";
ring r=0,(x),dp;
matrix A[5][4] = 1, 3,-1,4,
2, 5,-1,3,
1, 3,-1,4,
0, 4,-3,1,
-3,1,-5,-2;
list Z=gaussred_pivot(A);  //construct P,U,S s.t. P*A=U*S
print(Z[1]);               //P
print(Z[2]);               //U
print(Z[3]);               //S
print(Z[4]);               //rank
print(Z[1]*A);             //P*A
print(Z[2]*Z[3]);          //U*S
```

D.3.2.13 gauss_nf

Procedure from library `linalg.lib` (see Section D.3.2 [linalg_lib], page 353).

Usage: gauss_nf(A); A any constant matrix

Return: matrix; gauss normal form of A (uses gaussred)

Example:

```
LIB "linalg.lib";
ring r = 0,(x),dp;
matrix A[4][4] = 1,4,4,7,2,5,5,4,4,1,1,3,0,2,2,7;
print(gauss_nf(A));
```

D.3.2.14 mat_rk

Procedure from library `linalg.lib` (see Section D.3.2 [linalg_lib], page 353).

Usage: mat_rk(A); A any constant matrix

Return: int, rank of A

Example:

```
LIB "linalg.lib";
ring r = 0,(x),dp;
matrix A[4][4] = 1,4,4,7,2,5,5,4,4,1,1,3,0,2,2,7;
mat_rk(A);
```

D.3.2.15 U_D_O

Procedure from library `linalg.lib` (see Section D.3.2 [linalg_lib], page 353).

Usage: U_D_O(A); constant invertible matrix A

Return: list Z: Z[1]=P , Z[2]=U , Z[3]=D , Z[4]=O
gives a permutation matrix P,
a normalized lower triangular matrix U ,
a diagonal matrix D, and
a normalized upper triangular matrix O
with P*A=U*D*O

Note: Z[1]=-1 means that A is not regular (proc uses gaussred)

Example:

```
LIB "linalg.lib";
ring r = 0,(x),dp;
matrix A[5][5] = 10, 4,  0, -9,  8,
-3, 6, -6, -4,  9,
0, 3, -1, -9, -8,
-4,-2, -6, -10,10,
-9, 5, -1, -6,  5;
list Z = U_D_O(A);              //construct P,U,D,O s.t. P*A=U*D*O
print(Z[1]);                    //P
print(Z[2]);                    //U
print(Z[3]);                    //D
print(Z[4]);                    //O
print(Z[1]*A);                  //P*A
print(Z[2]*Z[3]*Z[4]);          //U*D*O
```

D.3.2.16 pos_def

Procedure from library `linalg.lib` (see Section D.3.2 [linalg_lib], page 353).

Usage: pos_def(A); A = constant, symmetric square matrix

Return: int:
1 if A is positive definit ,
0 if not,
-1 if unknown

Example:

```
LIB "linalg.lib";
ring r = 0,(x),dp;
matrix A[5][5] = 20,  4,  0, -9,   8,
4, 12, -6, -4,   9,
0, -6, -2, -9,  -8,
-9, -4, -9, -20, 10,
8,  9, -8,  10, 10;
pos_def(A);
matrix B[3][3] =  3,  2,  0,
2, 12,  4,
0,  4,  2;
pos_def(B);
```

D.3.2.17 hessenberg

Procedure from library `linalg.lib` (see Section D.3.2 [linalg_lib], page 353).

Usage: hessenberg(M); matrix M

Assume: M constant square matrix

Return: matrix H; Hessenberg form of M

Example:

```
LIB "linalg.lib";
ring R=0,x,dp;
matrix M[3][3]=3,2,1,0,2,1,0,0,3;
print(M);
print(hessenberg(M));
```

D.3.2.18 evnf

Procedure from library `linalg.lib` (see Section D.3.2 [linalg_lib], page 353).

Usage: evnf(e[,m]); ideal e, intvec m

Assume: ncols(e)==size(m)

Return: order eigenvalues e with multiplicities m

D.3.2.19 eigenvals

Procedure from library `linalg.lib` (see Section D.3.2 [linalg_lib], page 353).

Usage: eigenvals(M); matrix M

Assume: eigenvalues of M in basefield

Return:

list l;
  ideal l[1];
    number l[1][i]; i-th eigenvalue of M
  intvec l[2];
    int l[2][i]; multiplicity of i-th eigenvalue of M

Example:

```
LIB "linalg.lib";
ring R=0,x,dp;
matrix M[3][3]=3,2,1,0,2,1,0,0,3;
print(M);
eigenvals(M);
```

D.3.2.20 jordan

Procedure from library `linalg.lib` (see Section D.3.2 [linalg_lib], page 353).

Usage: jordan(M); matrix M

Assume: eigenvalues of M in basefield

Return:

list l; Jordan data of M
  ideal l[1];
    number l[1][i]; eigenvalue of i-th Jordan block of M
  intvec l[2];
    int l[2][i]; size of i-th Jordan block of M
  intvec l[3];
    int l[3][i]; multiplicity of i-th Jordan block of M

Example:

```
LIB "linalg.lib";
ring R=0,x,dp;
matrix M[3][3]=3,2,1,0,2,1,0,0,3;
print(M);
jordan(M);
```

D.3.2.21 jordanbasis

Procedure from library `linalg.lib` (see Section D.3.2 [linalg_lib], page 353).

Usage: jordanbasis(M); matrix M

Assume: eigenvalues of M in basefield

Return:

list l:
  module l[1]; inverse(l[1])*M*l[1] in Jordan normal form
  intvec l[2];
    int l[2][i]; weight filtration index of l[1][i]

Example:

```
LIB "linalg.lib";
ring R=0,x,dp;
matrix M[3][3]=3,2,1,0,2,1,0,0,3;
```

```
print(M);
list l=jordanbasis(M);
print(l[1]);
print(l[2]);
print(inverse(l[1])*M*l[1]);
```

D.3.2.22 jordanmatrix

Procedure from library `linalg.lib` (see Section D.3.2 [linalg_lib], page 353).

Usage: jordanmatrix(e,s,m); ideal e, intvec s, intvec m

Assume: ncols(e)==size(s)==size(m)

Return:

matrix J; Jordan matrix with list(e,s,m)==jordan(J)

Example:

```
LIB "linalg.lib";
ring R=0,x,dp;
ideal e=ideal(2,3);
intvec s=1,2;
intvec m=1,1;
print(jordanmatrix(e,s,m));
```

D.3.2.23 jordannf

Procedure from library `linalg.lib` (see Section D.3.2 [linalg_lib], page 353).

Usage: jordannf(M); matrix M

Assume: eigenvalues of M in basefield

Return: matrix J; Jordan normal form of M

Example:

```
LIB "linalg.lib";
ring R=0,x,dp;
matrix M[3][3]=3,2,1,0,2,1,0,0,3;
print(M);
print(jordannf(M));
```

D.4 Commutative algebra

D.4.1 algebra_lib

Library: algebra.lib

Purpose: Compute with Algebras and Algebra Maps

Authors: Gert-Martin Greuel, greuel@mathematik.uni-kl.de,
Agnes Eileen Heydtmann, agnes@math.uni-sb.de,
Gerhard Pfister, pfister@mathematik.uni-kl.de

Procedures:

D.4.1.1 algebra_containment

Procedure from library `algebra.lib` (see Section D.4.1 [algebra_lib], page 361).

Usage: algebra_containment(p,A[,k]); p poly, A ideal, k integer.
A = A[1],...,A[m] generators of subalgebra of the basering

Return:

- k=0 (or if k is not given) an integer:
  1 : if p is contained in the subalgebra K[A[1],...,A[m]]
  0 : if p is not contained in K[A[1],...,A[m]]
- k=1 : a list, say l, of size 2, l[1] integer, l[2] ring, satisfying
  l[1]=1 if p is in the subalgebra K[A[1],...,A[m]] and then the ring
  l[2] contains poly check = h(y(1),...,y(m)) if p=h(A[1],...,A[m])
  l[1]=0 if p is in not the subalgebra K[A[1],...,A[m]] and then
  l[2] contains the poly check = h(x,y(1),...,y(m)) if p satisfies
  the nonlinear relation p = h(x,A[1],...,A[m]) where
  x = x(1),...,x(n) denote the variables of the basering

Display: if k=0 and printlevel >= voice+1 (default) display the poly check

Note: The proc inSubring uses a different algorithm which is sometimes faster.

Theory: The ideal of algebraic relations of the algebra generators A[1],..., A[m] is computed introducing new variables y(i) and the product order with x(i) >> y(i).
p reduces to a polynomial only in the y(i) <=> p is contained in the subring generated by the polynomials A[1],...,A[m].

Example:

```
LIB "algebra.lib";
int p = printlevel; printlevel = 1;
ring R = 0,(x,y,z),dp;
ideal A=x2+y2,z2,x4+y4,1,x2z-1y2z,xyz,x3y-1xy3;
poly p1=z;
poly p2=
x10z3-x8y2z3+2x6y4z3-2x4y6z3+x2y8z3-y10z3+x6z4+3x4y2z4+3x2y4z4+y6z4;
algebra_containment(p1,A);
↦ // x(3)
↦ 0
algebra_containment(p2,A);
↦ // y(1)*y(2)*y(5)^2+y(3)*y(5)^3+4*y(1)*y(2)*y(6)^2+4*y(6)^3*y(7)+2*y(2)*y\
   (5)*y(7)^2
↦ 1
list L = algebra_containment(p2,A,1);
↦
↦ // 'algebra_containment' created a ring as 2nd element of the list.
↦ // The ring contains the poly check which defines the algebraic relation.
↦ // To access to the ring and see check you must give the ring a name,
↦ // e.g.:
↦                def S = l[2]; setring S; check;
↦
L[1];
↦ 1
def S = L[2]; setring S;
check;
↦ y(1)*y(2)*y(5)^2+y(3)*y(5)^3+4*y(1)*y(2)*y(6)^2+4*y(6)^3*y(7)+2*y(2)*y(5)\
```

```
    *y(7)^2
printlevel = p;
```

D.4.1.2 module_containment

Procedure from library `algebra.lib` (see Section D.4.1 [algebra_lib], page 361).

Usage: module_containment(p,P,M[,k]); p poly, P ideal, M ideal, k int
P = P[1],...,P[n] generators of a subalgebra of the basering,
M = M[1],...,M[m] generators of a module over the subalgebra K[P]

Assume: ncols(P) = nvars(basering), the P[i] are algebraically independent

Return:

- k=0 (or if k is not given), an integer:
  1 : if p is contained in the module <M[1],...,M[m]> over K[P]
  0 : if p is not contained in <M[1],...,M[m]>
- k=1, a list, say l, of size 2, l[1] integer, l[2] ring:
  l[1]=1 : if p is in <M[1],...,M[m]> and then the ring l[2] contains
  the polynomial check = h(y(1),...,y(m),z(1),...,z(n)) if
  p = h(M[1],...,M[m],P[1],...,P[n])
  l[1]=0 : if p is in not in <M[1],...,M[m]>, then l[2] contains the
  poly check = h(x,y(1),...,y(m),z(1),...,z(n)) if p satisfies
  the nonlinear relation p = h(x,M[1],...,M[m],P[1],...,P[n]) where
  x = x(1),...,x(n) denote the variables of the basering

Display: the polynomial h(y(1),...,y(m),z(1),...,z(n)) if k=0, resp. a comment how to access the relation check if k=1, provided printlevel >= voice+1 (default).

Theory: The ideal of algebraic relations of all the generators p1,...,pn, s1,...,st given by P and S is computed introducing new variables y(j), z(i) and the product order: x^a*y^b*z^c > x^d*y^e*z^f if x^a > x^d with respect to the lp ordering or else if z^c > z^f with respect to the dp ordering or else if y^b > y^e with respect to the lp ordering again. p reduces to a polynomial only in the y(j) and z(i), linear in the z(i) <=> p is contained in the module.

Example:

```
LIB "algebra.lib";
int p = printlevel; printlevel = 1;
ring R=0,(x,y,z),dp;
ideal P = x2+y2,z2,x4+y4;           //algebra generators
ideal M = 1,x2z-1y2z,xyz,x3y-1xy3;  //module generators
poly p1=
x10z3-x8y2z3+2x6y4z3-2x4y6z3+x2y8z3-y10z3+x6z4+3x4y2z4+3x2y4z4+y6z4;
module_containment(p1,P,M);
↦ // y(2)*z(2)*z(3)^2+z(1)^3*z(2)^2
↦ 1
poly p2=z;
list l = module_containment(p2,P,M,1);
↦
↦ // 'module_containment' created a ring as 2nd element of the list. The
↦ // ring contains the poly check which defines the algebraic relation
↦ // for p. To access to the ring and see check you must give the ring
↦ // a name, e.g.:
↦      def S = l[2]; setring S; check;
```

```
↦
l[1];
↦ 0
def S = l[2]; setring S; check;
↦ x(3)
printlevel=p;
```

D.4.1.3 inSubring

Procedure from library `algebra.lib` (see Section D.4.1 [algebra_lib], page 361).

Usage: inSubring(p,i); p poly, i ideal

Return:

a list l of size 2, l[1] integer, l[2] string
l[1]=1 iff p is in the subring generated by i=i[1],...,i[k],
    and then l[2] = y(0)-h(y(1),...,y(k)) if p = h(i[1],...,i[k])
l[1]=0 iff p is in not the subring generated by i,
    and then l[2] = h(y(0),y(1),...,y(k) where p satisfies the
    nonlinear relation h(p,i[1],...,i[k])=0.

Note: the proc algebra_containment tests the same with a different algorithm, which is often faster

Example:

```
LIB "algebra.lib";
ring q=0,(x,y,z,u,v,w),dp;
poly p=xyzu2w-1yzu2w2+u4w2-1xu2vw+u2vw2+xyz-1yzw+2u2w-1xv+vw+2;
ideal I =x-w,u2w+1,yz-v;
inSubring(p,I);
↦ [1]:
↦    1
↦ [2]:
↦    y(1)*y(2)*y(3)+y(2)^2-y(0)+1
```

D.4.1.4 algDependent

Procedure from library `algebra.lib` (see Section D.4.1 [algebra_lib], page 361).

Usage: algDependent(f[,c]); f ideal (say, f = f1,...,fm), c integer

Return:

a list l of size 2, l[1] integer, l[2] ring:
- l[1] = 1 if f1,...,fm are algebraic dependent, 0 if not
- l[2] is a ring with variables x(1),...,x(n),y(1),...,y(m) if the
  basering has n variables. It contains the ideal 'ker', depending
  only on the y(i) and generating the algebraic relations between the
  f[i], i.e. substituting y(i) by fi yields 0. Of course, ker is
  nothing but the kernel of the ring map
    K[y(1),...,y(m)] ---> basering, y(i) --> fi.

Note: Three different algorithms are used depending on c = 1,2,3. If c is not given or c=0, a heuristically best method is chosen. The basering may be a quotient ring.
To access to the ring l[2] and see ker you must give the ring a name, e.g. def S=l[2]; setring S; ker;

Display: The above comment is displayed if printlevel >= 0 (default).

Example:

```
LIB "algebra.lib";
int p = printlevel; printlevel = 1;
ring R = 0,(x,y,z,u,v,w),dp;
ideal I = xyzu2w-1yzu2w2+u4w2-1xu2vw+u2vw2+xyz-1yzw+2u2w-1xv+vw+2,
x-w, u2w+1, yz-v;
list l = algDependent(I);
↦
↦ // The 2nd element of the list l is a ring with variables x(1),...,x(n),
↦ // and y(1),...,y(m) if the basering has n variables and if the ideal
↦ // is f[1],...,f[m]. The ring contains the ideal ker which depends only
↦ // on the y(i) and generates the relations between the f[i].
↦ // I.e. substituting y(i) by f[i] yields 0.
↦ // To access to the ring and see ker you must give the ring a name,
↦ // e.g.:
↦               def S = l[2]; setring S; ker;
↦
l[1];
↦ 1
def S = l[2]; setring S;
ker;
↦ ker[1]=y(2)*y(3)*y(4)+y(3)^2-y(1)+1
printlevel = p;
```

D.4.1.5 alg_kernel

Procedure from library `algebra.lib` (see Section D.4.1 [algebra_lib], page 361).

Usage: alg_kernel(phi,pr[,s,c]); phi map to basering, pr preimage ring, s string (name of kernel in pr), c integer.

Return: a string, the kernel of phi as string.
If, moreover, a string s is given, the algorithm creates, in the preimage ring pr the kernel of phi with name s.
Three different algorithms are used depending on c = 1,2,3. If c is not given or c=0, a heuristically best method is chosen. (algorithm 1 uses the preimage command)

Note: Since the kernel of phi lives in pr, it cannot be returned to the basering. If s is given, the user has access to it in pr via s. The basering may be a quotient ring.

Example:

```
LIB "algebra.lib";
ring r = 0,(a,b,c),ds;
ring s = 0,(x,y,z,u,v,w),dp;
ideal I = x-w,u2w+1,yz-v;
map phi = r,I;                // a map from r to s:
alg_kernel(phi,r);            // a,b,c ---> x-w,u2w+1,yz-v
↦ 0
ring S = 0,(a,b,c),ds;
ring R = 0,(x,y,z),dp;
qring Q = std(x-y);
ideal i = x, y, x2-y3;
map phi = S,i;                 // a map to a quotient ring
```

```
alg_kernel(phi,S,"ker",3);     // uses algorithm 3
↦ a-b,b^3-b^2+c
setring S;                     // you have access to kernel in preimage
ker;
↦ ker[1]=a-b
↦ ker[2]=c-b2+b3
```

D.4.1.6 is_injective

Procedure from library `algebra.lib` (see Section D.4.1 [algebra_lib], page 361).

Usage: is_injective(phi,pr[,c,s]); phi map, pr preimage ring, c int, s string

Return:

- 1 (type int) if phi is injective, 0 if not (if s is not given).
- If s is given, return a list l of size 2, l[1] int, l[2] ring:
  l[1] is 1 if phi is injective, 0 if not
  l[2] is a ring with variables x(1),...,x(n),y(1),...,y(m) if the basering has n variables and the map m components, it contains the ideal 'ker', depending only on the y(i), the kernel of the given map

Note: Three different algorithms are used depending on c = 1,2,3. If c is not given or c=0, a heuristically best method is chosen. The basering may be a quotient ring. However, if the preimage ring is a quotient ring, say pr = P/I, consider phi as a map from P and then the algorithm returns 1 if the kernel of phi is 0 mod I. To access to the ring l[2] and see ker you must give the ring a name, e.g. def S=l[2]; setring S; ker;

Display: The above comment is displayed if printlevel >= 0 (default).

Example:

```
LIB "algebra.lib";
int p = printlevel;
ring r = 0,(a,b,c),ds;
ring s = 0,(x,y,z,u,v,w),dp;
ideal I = x-w,u2w+1,yz-v;
map phi = r,I;                    // a map from r to s:
is_injective(phi,r);              // a,b,c ---> x-w,u2w+1,yz-v
↦ 1
ring R = 0,(x,y,z),dp;
ideal i = x, y, x2-y3;
map phi = R,i;                    // a map from R to itself, z --> x2-y3
list l = is_injective(phi,R,"");
↦
↦ // The 2nd element of the list is a ring with variables x(1),...,x(n),
↦ // y(1),...,y(m) if the basering has n variables and the map is
↦ // F[1],...,F[m].
↦ // It contains the ideal ker, the kernel of the given map y(i) --> F[i].
↦ // To access to the ring and see ker you must give the ring a name,
↦ // e.g.:
↦      def S = l[2]; setring S; ker;
↦
l[1];
↦ 0
def S = l[2]; setring S;
ker;
↦ ker[1]=y(2)^3-y(1)^2+y(3)
```

D.4.1.7 is_surjective

Procedure from library `algebra.lib` (see Section D.4.1 [algebra_lib], page 361).

Usage: is_surjective(phi); phi map to basering, or ideal defining it

Return: an integer, 1 if phi is surjective, 0 if not

Note: The algorithm returns 1 iff all the variables of the basering are contained in the polynomial subalgebra generated by the polynomials defining phi. Hence, if the basering has local or mixed ordering or if the preimage ring is a quotient ring (in which case the map may not be well defined) then the return value 1 means
"surjectivity" in this sense.

Example:

```
LIB "algebra.lib";
ring R = 0,(x,y,z),dp;
ideal i = x, y, x2-y3;
map phi = R,i;                     // a map from R to itself, z->x2-y3
is_surjective(phi);
↦ 0
qring Q = std(ideal(z-x37));
map psi = R, x,y,x2-y3;            // the same map to the quotient ring
is_surjective(psi);
↦ 1
ring S = 0,(a,b,c),dp;
map psi = R,ideal(a,a+b,c-a2+b3); // a map from R to S,
is_surjective(psi);                // x->a, y->a+b, z->c-a2+b3
↦ 1
```

D.4.1.8 is_bijective

Procedure from library `algebra.lib` (see Section D.4.1 [algebra_lib], page 361).

Usage: is_bijective(phi,pr); phi map to basering, pr preimage ring

Return: an integer, 1 if phi is bijective, 0 if not

Note: The algorithm checks first injectivity and then surjectivity To interpret this for local/mixed orderings, or for quotient rings type help is_surjective; and help is_injective;

Display: A comment if printlevel >= voice-1 (default)

Example:

```
LIB "algebra.lib";
int p = printlevel;  printlevel = 1;
ring R = 0,(x,y,z),dp;
ideal i = x, y, x2-y3;
map phi = R,i;                       // a map from R to itself, z->x2-y3
is_bijective(phi,R);
↦ // map not injective
↦ 0
qring Q = std(z-x2+y3);
is_bijective(ideal(x,y,x2-y3),Q);
↦ 1
ring S = 0,(a,b,c,d),dp;
map psi = R,ideal(a,a+b,c-a2+b3,0); // a map from R to S,
```

```
is_bijective(psi,R);                // x->a, y->a+b, z->c-a2+b3
↦ // map injective, but not surjective
↦ 0
qring T = std(d,c-a2+b3);
↦ // ** _ is no standardbasis
map chi = Q,a,b,a2-b3;              // amap between two quotient rings
is_bijective(chi,Q);
↦ 1
printlevel = p;
```

D.4.1.9 noetherNormal

Procedure from library `algebra.lib` (see Section D.4.1 [algebra_lib], page 361).

Usage: noetherNormal(id[,p]); id ideal, p integer

Return:

a list l two ideals, say I,J:
- I is generated by a subset of the variables with size(I) = dim(id)
- J defines a map (coordinate change in the basering), such that:
  if we define map phi=basering,J;
  then k[var(1),...,var(n)]/phi(id) is finite over k[I].

If p is given, 0<=p<=100, a sparse coordinate change with p percent of the matrix entries being 0 (default: p=0 i.e. dense)

Note: Designed for characteristic 0.It works also in char k > 0 if it terminates,but may result in an infinite loop in small characteristic

Example:

```
LIB "algebra.lib";
ring r=0,(x,y,z),dp;
ideal i= xy,xz;
noetherNormal(i);
↦ [1]:
↦    _[1]=x
↦    _[2]=2x+y
↦    _[3]=3x+4y+z
↦ [2]:
↦    _[1]=y
↦    _[2]=z
```

D.4.1.10 mapIsFinite

Procedure from library `algebra.lib` (see Section D.4.1 [algebra_lib], page 361).

Usage: mapIsFinite(phi,R[,J]); R a ring, phi: R —> basering a map J an ideal in the basering, J = 0 if not given

Return: 1 if R —> basering/J is finite and 0 else

Example:

```
LIB "algebra.lib";
ring r = 0,(a,b,c),dp;
ring s = 0,(x,y,z),dp;
ideal i= xy;
map phi= r,(xy)^3+x2+z,y2-1,z3;
```

```
mapIsFinite(phi,r,i);
↦ 1
```

D.4.1.11 finitenessTest

Procedure from library `algebra.lib` (see Section D.4.1 [algebra_lib], page 361).

Usage: finitenessTest(J[,v]); J ideal, v intvec (say v1,...,vr with vi>0)

Return:

a list l with l[1] integer, l[2], l[3], l[4] ideals
- l[1] = 1 if var(v1),...,var(vr) are in l[2] and 0 else
- l[2] (resp. l[3]) contains those variables which occur,
  (resp. occur not) as pure power in the leading term of one of the generators of J,
- l[4] contains those J[i] for which the leading term is a pure power
  of a variable (which is then in l[2])

(default: v = [1,2,..,nvars(basering)])

Theory: If J is a standard basis of an ideal generated by x_1 - f_1(y),..., x_n - f_n with y_j ordered lexicographically and y_j >> x_i, then, if y_i appears as pure power in the leading term of J[k]. J[k] defines an integral relation for y_i over the y_(i+1),... and the f's. Moreover, in this situation, if l[2] = y_1,...,y_r, then K[y_1,...y_r] is finite over K[f_1..f_n]. If J contains furthermore polynomials h_j(y), then K[y_1,...y_z]/<h_j> is finite over K[f_1..f_n].

Example:

```
LIB "algebra.lib";
ring s = 0,(x,y,z,a,b,c),(lp(3),dp);
ideal i= a -(xy)^3+x2-z, b -y2-1, c -z3;
ideal j = a -(xy)^3+x2-z, b -y2-1, c -z3, xy;
finitenessTest(std(i),1..3);
↦ [1]:
↦    0
↦ [2]:
↦    _[1]=y
↦    _[2]=z
↦ [3]:
↦    _[1]=x
↦    _[2]=a
↦    _[3]=b
↦    _[4]=c
↦ [4]:
↦    _[1]=z3-c
↦    _[2]=y2-b+1
finitenessTest(std(j),1..3);
↦ [1]:
↦    1
↦ [2]:
↦    _[1]=x
↦    _[2]=y
↦    _[3]=z
↦ [3]:
↦    _[1]=a
↦    _[2]=b
```

```
↦    _[3]=c
↦ [4]:
↦    _[1]=z3-c
↦    _[2]=y2-b+1
↦    _[3]=x2-z+a
```

D.4.2 elim_lib

Library: elim.lib

Purpose: Elimination, Saturation and Blowing up

Procedures:

D.4.2.1 blowup0

Procedure from library `elim.lib` (see Section D.4.2 [elim_lib], page 370).

Usage: blowup0(j[,s1,s2]); j ideal, s1,s2 nonempty strings

Create: Create a presentation of the blowup ring of j

Return: no return value

Note: s1 and s2 are used to give names to the blownup ring and the blownup ideal (default: s1="j", s2="A")
Assume R = char,x(1..n),ord is the basering of j, and s1="j", s2="A" then the procedure creates a new ring with name Bl_jR
(equal to R[A,B,...])
Bl_jR = char,(A,B,...,x(1..n)),(dp(k),ord)
with k=ncols(j) new variables A,B,... and ordering wp(d1..dk) if j is homogeneous with deg(j[i])=di resp. dp otherwise for these vars. If k>26 or size(s2)>1, say s2="A()", the new vars are A(1),...,A(k). Let j_ be the kernel of the ring map Bl_jR -> R defined by A(i)->j[i], x(i)->x(i), then the quotient ring Bl_jR/j_ is the blowup ring of j in R (being isomorphic to R+j+j^2+...). Moreover the procedure creates a std basis of j_ with name j_ in Bl_jR.
This proc uses 'execute' or calls a procedure using 'execute'.

Display: printlevel >=0: explain created objects (default)

Example:

```
LIB "elim.lib";
ring R=0,(x,y),dp;
poly f=y2+x3; ideal j=jacob(f);
blowup0(j);
↦
↦ // The proc created the ring Bl_jR (equal to R[A,B])
↦ // it contains the ideal j_ , such that
↦ //             Bl_jR/j_ is the blowup ring
↦ // show(Bl_jR); shows this ring.
↦ // Make Bl_jR the basering and see j_ by typing:
↦    setring Bl_jR;
↦    j_;
show(Bl_jR);
↦ // ring: (0),(A,B,x,y),(wp(2,1),dp(2),C);
↦ // minpoly = 0
```

```
↦ // objects belonging to this ring:
↦ // j_                  [0]  ideal, 1 generator(s)
setring Bl_jR;
j_;"";
↦ j_[1]=2Ay-3Bx2
↦
ring r=32003,(x,y,z),ds;
blowup0(maxideal(1),"m","T()");
↦
↦ // The proc created the ring Bl_mr (equal to r[T(1..3)])
↦ // it contains the ideal m_ , such that
↦ //             Bl_mr/m_ is the blowup ring
↦ // show(Bl_mr); shows this ring.
↦ // Make Bl_mr the basering and see m_ by typing:
↦    setring Bl_mr;
↦    m_;
show(Bl_mr);
↦ // ring: (32003),(T(1),T(2),T(3),x,y,z),(wp(1,1,1),ds(3),C);
↦ // minpoly = 0
↦ // objects belonging to this ring:
↦ // m_                  [0]  ideal, 3 generator(s)
setring Bl_mr;
m_;
↦ m_[1]=T(1)y-T(2)x
↦ m_[2]=T(1)z-T(3)x
↦ m_[3]=T(2)z-T(3)y
kill Bl_jR, Bl_mr;
```

D.4.2.2 elim

Procedure from library `elim.lib` (see Section D.4.2 [elim_lib], page 370).

Usage: elim(id,n,m); id ideal/module, n,m integers

Returns: ideal/module obtained from id by eliminating variables n..m

Note: no special monomial ordering is required, result is a SB with respect to ordering dp (resp. ls) if the first var not to be eliminated belongs to a -p (resp. -s) block ordering This proc uses 'execute' or calls a procedure using 'execute'.

Example:

```
LIB "elim.lib";
ring r=0,(x,y,u,v,w),dp;
ideal i=x-u,y-u2,w-u3,v-x+y3;
elim(i,3,4);
↦ _[1]=y2-xw
↦ _[2]=xy-w
↦ _[3]=x2-y
module m=i*gen(1)+i*gen(2);
m=elim(m,3,4);show(m);
↦ // module, 6 generator(s)
↦ [y2-xw]
↦ [0,y2-xw]
↦ [xy-w]
↦ [0,xy-w]
```

```
↦ [x2-y]
↦ [0,x2-y]
```

See also: Section D.4.2.3 [elim1], page 372; Section 5.1.22 [eliminate], page 136.

D.4.2.3 elim1

Procedure from library `elim.lib` (see Section D.4.2 [elim_lib], page 370).

Usage: elim1(id,p); id ideal/module, p product of vars to be eliminated

Return: ideal/module obtained from id by eliminating vars occurring in poly

Note: no special monomial ordering is required, result is a SB with respect to ordering dp (resp. ls) if the first var not to be eliminated belongs to a -p (resp. -s) block ordering This proc uses 'execute' or calls a procedure using 'execute'.

Example:

```
LIB "elim.lib";
ring r=0,(x,y,t,s,z),dp;
ideal i=x-t,y-t2,z-t3,s-x+y3;
elim1(i,ts);
↦ _[1]=y2-xz
↦ _[2]=xy-z
↦ _[3]=x2-y
module m=i*gen(1)+i*gen(2);
m=elim1(m,st); show(m);
↦ // module, 6 generator(s)
↦ [y2-xz]
↦ [0,y2-xz]
↦ [xy-z]
↦ [0,xy-z]
↦ [x2-y]
↦ [0,x2-y]
```

See also: Section D.4.2.2 [elim], page 371; Section 5.1.22 [eliminate], page 136.

D.4.2.4 nselect

Procedure from library `elim.lib` (see Section D.4.2 [elim_lib], page 370).

Usage: nselect(id,n[,m]); id a module or ideal, n, m integers

Return: generators of id not containing the variable n [up to m]

Example:

```
LIB "elim.lib";
ring r=0,(x,y,t,s,z),(c,dp);
ideal i=x-y,y-z2,z-t3,s-x+y3;
nselect(i,3);
↦ _[1]=x-y
↦ _[2]=-z2+y
↦ _[3]=y3-x+s
module m=i*(gen(1)+gen(2));
show(m);
↦ // module, 4 generator(s)
↦ [x-y,x-y]
```

```
↦ [-z2+y,-z2+y]
↦ [-t3+z,-t3+z]
↦ [y3-x+s,y3-x+s]
show(nselect(m,3,4));
↦ // module, 2 generator(s)
↦ [x-y,x-y]
↦ [-z2+y,-z2+y]
```

See also: Section D.4.2.6 [select], page 373; Section D.4.2.7 [select1], page 374.

D.4.2.5 sat

Procedure from library `elim.lib` (see Section D.4.2 [elim_lib], page 370).

Usage: sat(id,j); id=ideal/module, j=ideal

Return: list of an ideal/module [1] and an integer [2]:
[1] = saturation of id with respect to j (= union_(k=1...) of id:j^k) [2] = saturation exponent (= min(k | id:j^k = id:j^(k+1)))

Note: [1] is a standard basis in the basering

Display: saturation exponent during computation if printlevel >=1

Example:

```
LIB "elim.lib";
int p      = printlevel;
ring r     = 2,(x,y,z),dp;
poly F     = x5+y5+(x-y)^2*xyz;
ideal j    = jacob(F);
sat(j,maxideal(1));
↦ [1]:
↦    _[1]=x3+x2y+xy2+y3
↦    _[2]=y4+x2yz+y3z
↦    _[3]=x2y2+x2yz+y3z
↦ [2]:
↦    4
printlevel = 2;
sat(j,maxideal(2));
↦ // compute quotient 1
↦ // compute quotient 2
↦ // compute quotient 3
↦ // saturation becomes stable after 2 iteration(s)
↦
↦ [1]:
↦    _[1]=x3+x2y+xy2+y3
↦    _[2]=y4+x2yz+y3z
↦    _[3]=x2y2+x2yz+y3z
↦ [2]:
↦    2
printlevel = p;
```

D.4.2.6 select

Procedure from library `elim.lib` (see Section D.4.2 [elim_lib], page 370).

Usage: select(id,n[,m]); id ideal/module, n, m integers

Return: generators of id containing the variable n [all variables up to m]

Note: use 'select1' for selecting generators containing at least one of the variables between n and m

Example:

```
LIB "elim.lib";
ring r=0,(x,y,t,s,z),(c,dp);
ideal i=x-y,y-z2,z-t3,s-x+y3;
ideal j=select(i,1);
j;
↦ j[1]=x-y
↦ j[2]=y3-x+s
module m=i*(gen(1)+gen(2));
m;
↦ m[1]=[x-y,x-y]
↦ m[2]=[-z2+y,-z2+y]
↦ m[3]=[-t3+z,-t3+z]
↦ m[4]=[y3-x+s,y3-x+s]
select(m,1,2);
↦ _[1]=[x-y,x-y]
↦ _[2]=[y3-x+s,y3-x+s]
```

See also: Section D.4.2.4 [nselect], page 372; Section D.4.2.7 [select1], page 374.

D.4.2.7 select1

Procedure from library `elim.lib` (see Section D.4.2 [elim_lib], page 370).

Usage: select1(id,n[,m]); id ideal/module, n, m integers

Return: generators of id containing the variable n
[at least one of the variables up to m]

Note: use 'select' for selecting generators containing all the variables between n and m

Example:

```
LIB "elim.lib";
ring r=0,(x,y,t,s,z),(c,dp);
ideal i=x-y,y-z2,z-t3,s-x+y3;
ideal j=select1(i,1);
j;
↦ j[1]=x-y
↦ j[2]=y3-x+s
module m=i*(gen(1)+gen(2));
m;
↦ m[1]=[x-y,x-y]
↦ m[2]=[-z2+y,-z2+y]
↦ m[3]=[-t3+z,-t3+z]
↦ m[4]=[y3-x+s,y3-x+s]
select1(m,1,2);
↦ _[1]=[x-y,x-y]
↦ _[2]=[-z2+y,-z2+y]
↦ _[3]=[y3-x+s,y3-x+s]
```

See also: Section D.4.2.4 [nselect], page 372; Section D.4.2.6 [select], page 373.

D.4.3 homolog_lib

Library: homolog.lib

Purpose: Procedures for Homological Algebra

Authors: Gert-Martin Greuel, greuel@mathematik.uni-kl.de,
Bernd Martin, martin@math.tu-cottbus.de

Procedures:

D.4.3.1 cup

Procedure from library `homolog.lib` (see Section D.4.3 [homolog_lib], page 375).

Usage: cup(M,[,any,any]); M=module

Compute: cup-product Ext^1(M',M') x Ext^1(M',M') —> Ext^2(M',M'), where M':=R^m/M, if M in R^m, R basering (i.e. M':=coker(matrix(M))).
If called with >= 2 arguments: compute symmetrized cup-product

Assume: all Ext's are finite dimensional

Return:

matrix of the associated linear map,
i.e. the columns of <matrix> present the coordinates of b_i & b_j
(resp. (1/2)(b_i & b_j + b_j & b_i) in the symmetric version)
with respect to a kbase of Ext^2,
where b_1,b_2,... is a kbase of Ext^1 and & denotes cup product.
In case of a third argument return a list:
L[1] = matrix see above (symmetric case, for >=2 arguments)
L[2] = matrix of kbase of Ext^1
L[3] = matrix of kbase of Ext^2

Note: printlevel >=1; shows what is going on.
printlevel >=2; shows result in another representation.
For computing cupproduct of M itself, apply proc to syz(M)!

Example:

```
LIB "homolog.lib";
int p      = printlevel;
ring  rr   = 32003,(x,y,z),(dp,C);
ideal  I   = x4+y3+z2;
qring  o   = std(I);
module M   = [x,y,0,z],[y2,-x3,z,0],[z,0,-y,-x3],[0,z,x,-y2];
print(cup(M));
↦ 0,1,0, 0, 0,0,0,0,0,0,0, 0,0,0,0,0,0,
↦ 0,0,-1,0, 0,1,0,0,0,0,0, 0,0,0,0,0,0,
↦ 0,0,0, -1,0,0,0,0,0,1,0, 0,0,0,0,0,0,
↦ 0,0,0, 0, 1,0,0,1,0,0,-1,0,0,1,0,0,0
print(cup(M,1));
↦ 0,1,0,0,0,0,0,0,0,0,0,0,
↦ 0,0,0,0,0,0,0,0,0,0,0,0,
↦ 0,0,0,0,0,0,0,0,0,0,0,0,
↦ 0,0,0,0,1,0,0,0,0,0,0,0
// 2nd EXAMPLE  (shows what is going on)
printlevel = 3;
```

```
ring   r   = 0,(x,y),(dp,C);
ideal  i   = x2-y3;
qring  q   = std(i);
module M   = [-x,y],[-y2,x];
print(cup(M));
↦ // vdim (Ext^1) = 2
↦ // kbase of Ext^1(M,M)
↦ //  - the columns present the kbase elements in Hom(F(1),F(0))
↦ //  - F(*) a free resolution of M
↦ -1,0,
↦ 0, y,
↦ 0, 1,
↦ -1,0
↦ // lift kbase of Ext^1:
↦ //  - the columns present liftings of kbase elements into Hom(F(2),F(1))
↦ //  - F(*) a free resolution of M
↦ 1,0,
↦ 0,y,
↦ 0,1,
↦ 1,0
↦ // vdim (Ext^2) = 2
↦ // kbase of Ext^2(M,M)
↦ //  - the columns present the kbase elements in Hom(F(2),F(0))
↦ //  - F(*) is a a free resolution of M
↦ -1,0,
↦ 0, y,
↦ 0, 1,
↦ -1,0
↦ // matrix of cup-products (in Ext^2)
↦ 0,-1,0, 0,y,
↦ 0,0, -y,y,0,
↦ 0,0, -1,1,0,
↦ 0,-1,0, 0,y
↦ ////// end level 2 //////
↦ // the associated matrices of the bilinear mapping 'cup'
↦ // corresponding to the kbase elements of Ext^2(M,M) are shown,
↦ //  i.e. the rows of the final matrix are written as matrix of
↦ //  a bilinear form on Ext^1 x Ext^1
↦ //_____1. component:
↦ 0,1,
↦ 0,0
↦ //_____2. component:
↦ 0, 0,
↦ -1,1
↦ ////// end level 3 //////
↦ 0,1,0, 0,0,
↦ 0,0,-1,1,0
printlevel = p;
```

D.4.3.2 cupproduct

Procedure from library `homolog.lib` (see Section D.4.3 [homolog_lib], page 375).

Usage: cupproduct(M,N,P,p,q[,any]); M,N,P modules, p,q integers

Compute: cup-product Ext^p(M',N') x Ext^q(N',P') —> Ext^p+q(M',P'), where M':=R^m/M, if M in R^m, R basering (i.e. M':=coker(matrix(M)))

Assume: all Ext's are of finite dimension

Return:

matrix of the associated linear map Ext^p(tensor)Ext^q–>Ext^p+q
i.e. the columns of <matrix> present the coordinates of
the cup products (b_i & c_j) with respect to a kbase of Ext^p+q
(b_i resp. c_j are choosen bases of Ext^p resp. Ext^q).
In case of a 6th argument:
return a list
L[1] = matrix (see above)
L[2] = matrix of kbase of Ext^p(M',N')
L[3] = matrix of kbase of Ext^q(N',P')
L[4] = matrix of kbase of Ext^p+q(N',P')

Note: printlevel >=1; shows what is going on
printlevel >=2; shows result in another representation For computing cupproduct of M,N itself, apply proc to syz(M),syz(N)!

Example:

```
LIB "homolog.lib";
int p      = printlevel;
ring  rr   = 32003,(x,y,z),(dp,C);
ideal  I   = x4+y3+z2;
qring  o   = std(I);
module M   = [x,y,0,z],[y2,-x3,z,0],[z,0,-y,-x3],[0,z,x,-y2];
print(cupproduct(M,M,M,1,3));
↦ 0,1,0, 0, 0,0,0,0,0,0,0, 0,0,0,0,0,0,
↦ 0,0,-1,0, 0,1,0,0,0,0,0, 0,0,0,0,0,0,
↦ 0,0,0, -1,0,0,0,0,0,1,0, 0,0,0,0,0,0,
↦ 0,0,0, 0, 1,0,0,1,0,0,-1,0,0,1,0,0,0
printlevel = 3;
list l     = (cupproduct(M,M,M,1,3,"any"));
↦ // vdim Ext(M,N) = 4
↦ // kbase of Ext^p(M,N)
↦ //  - the columns present the kbase elements in Hom(F(p),G(0))
↦ //  - F(*),G(*) are free resolutions of M and N
↦ 0, 0, 1, 0,
↦ 0, y, 0, 0,
↦ 1, 0, 0, 0,
↦ 0, 0, 0, y,
↦ 0, -1,0, 0,
↦ 0, 0, x2,0,
↦ 0, 0, 0, -x2,
↦ 1, 0, 0, 0,
↦ 0, 0, 0, -1,
↦ -1,0, 0, 0,
↦ 0, 1, 0, 0,
↦ 0, 0, 1, 0,
↦ -1,0, 0, 0,
↦ 0, 0, 0, x2y,
↦ 0, 0, x2,0,
↦ 0, -y,0, 0
```

```
↦ // vdim Ext(N,P) = 4
↦ // kbase of Ext(N,P):
↦ 0, 0, 1,  0,
↦ 0, 0, 0,  y,
↦ 1, 0, 0,  0,
↦ 0, -y,0,  0,
↦ 0, -1,0,  0,
↦ 1, 0, 0,  0,
↦ 0, 0, 0,  -x2,
↦ 0, 0, -x2,0,
↦ 0, 0, 0,  -1,
↦ 0, 0, 1,  0,
↦ 0, 1, 0,  0,
↦ 1, 0, 0,  0,
↦ -1,0, 0,  0,
↦ 0, -y,0,  0,
↦ 0, 0, x2, 0,
↦ 0, 0, 0,  -x2y
↦ // kbase of Ext^q(N,P)
↦ //  - the columns present the kbase elements in Hom(G(q),H(0))
↦ //  - G(*),H(*) are free resolutions of N and P
↦ 0, 0, 1,  0,
↦ 0, 0, 0,  y,
↦ 1, 0, 0,  0,
↦ 0, -y,0,  0,
↦ 0, -1,0,  0,
↦ 1, 0, 0,  0,
↦ 0, 0, 0,  -x2,
↦ 0, 0, -x2,0,
↦ 0, 0, 0,  -1,
↦ 0, 0, 1,  0,
↦ 0, 1, 0,  0,
↦ 1, 0, 0,  0,
↦ -1,0, 0,  0,
↦ 0, -y,0,  0,
↦ 0, 0, x2, 0,
↦ 0, 0, 0,  -x2y
↦ // vdim Ext(M,P) = 4
↦ // kbase of Ext^p+q(M,P)
↦ //  - the columns present the kbase elements in Hom(F(p+q),H(0))
↦ //  - F(*),H(*) are free resolutions of M and P
↦ 0, 0, 1,  0,
↦ 0, 0, 0,  y,
↦ 1, 0, 0,  0,
↦ 0, -y,0,  0,
↦ 0, -1,0,  0,
↦ 1, 0, 0,  0,
↦ 0, 0, 0,  -x2,
↦ 0, 0, -x2,0,
↦ 0, 0, 0,  -1,
↦ 0, 0, 1,  0,
↦ 0, 1, 0,  0,
↦ 1, 0, 0,  0,
```

```
↦ -1,0, 0,  0,
↦ 0, -y,0,  0,
↦ 0, 0, x2, 0,
↦ 0, 0, 0,  -x2y
↦ // lifting of kbase of Ext^p(M,N)
↦ //  - the columns present the lifting of kbase elements in Hom(F(p+q),G(q\
  ))
↦ 1,0, 0, 0,
↦ 0,-y,0, 0,
↦ 0,0, x2,0,
↦ 0,0, 0, x2y,
↦ 0,1, 0, 0,
↦ 1,0, 0, 0,
↦ 0,0, 0, -x2,
↦ 0,0, x2,0,
↦ 0,0, -1,0,
↦ 0,0, 0, y,
↦ 1,0, 0, 0,
↦ 0,y, 0, 0,
↦ 0,0, 0, -1,
↦ 0,0, -1,0,
↦ 0,-1,0, 0,
↦ 1,0, 0, 0
↦ // matrix of cup-products (in Ext^p+q)
↦ 0,0, 0, -1, 0,   0, 0, 0,   y,   1,  0,  0,  0,  0,   y,   0,  0,
↦ 0,0, 0, 0,  y,   0, 0, y,   0,   0,  -y, 0,  0,  y,   0,   0,  0,
↦ 0,1, 0, 0,  0,   0, y, 0,   0,   0,  0,  x2, 0,  0,   0,   0,  -x2y,
↦ 0,0, y, 0,  0,   -y,0, 0,   0,   0,  0,  0,  x2y,0,   0,   x2y,0,
↦ 0,0, 1, 0,  0,   -1,0, 0,   0,   0,  0,  0,  x2, 0,   0,   x2, 0,
↦ 0,1, 0, 0,  0,   0, y, 0,   0,   0,  0,  x2, 0,  0,   0,   0,  -x2y,
↦ 0,0, 0, 0,  -x2, 0, 0, -x2, 0,   0,  x2, 0,  0,  -x2, 0,   0,  0,
↦ 0,0, 0, x2, 0,   0, 0, 0,   -x2y,-x2,0,  0,  0,  0,   -x2y,0,  0,
↦ 0,0, 0, 0,  -1,  0, 0, -1,  0,   0,  1,  0,  0,  -1,  0,   0,  0,
↦ 0,0, 0, -1, 0,   0, 0, 0,   y,   1,  0,  0,  0,  0,   y,   0,  0,
↦ 0,0, -1,0,  0,   1, 0, 0,   0,   0,  0,  0,  -x2,0,   0,   -x2,0,
↦ 0,1, 0, 0,  0,   0, y, 0,   0,   0,  0,  x2, 0,  0,   0,   0,  -x2y,
↦ 0,-1,0, 0,  0,   0, -y,0,   0,   0,  0,  -x2,0,  0,   0,   0,  x2y,
↦ 0,0, y, 0,  0,   -y,0, 0,   0,   0,  0,  0,  x2y,0,   0,   x2y,0,
↦ 0,0, 0, -x2,0,   0, 0, 0,   x2y, x2, 0,  0,  0,  0,   x2y, 0,  0,
↦ 0,0, 0, 0,  -x2y,0, 0, -x2y,0,   0,  x2y,0,  0,  -x2y,0,   0,  0
↦ ////// end level 2 //////
↦ // the associated matrices of the bilinear mapping 'cup'
↦ // corresponding to the kbase elements of Ext^p+q(M,P) are shown,
↦ //  i.e. the rows of the final matrix are written as matrix of
↦ //  a bilinear form on Ext^p x Ext^q
↦ //----1. component:
↦ 0,1,0,0,
↦ 0,0,0,0,
↦ 0,0,0,0,
↦ 0,0,0,0
↦ //----2. component:
↦ 0,0,-1,0,
↦ 0,1,0, 0,
```

```
↦ 0,0,0, 0,
↦ 0,0,0, 0
↦ //----3. component:
↦ 0,0,0,-1,
↦ 0,0,0,0,
↦ 0,1,0,0,
↦ 0,0,0,0
↦ //----4. component:
↦ 0,0,0, 0,
↦ 1,0,0, 1,
↦ 0,0,-1,0,
↦ 0,1,0, 0
↦ ////// end level 3 //////
show(l[1]);show(l[2]);
↦ // matrix, 4x17
↦ 0,1,0, 0, 0,0,0,0,0,0,0, 0,0,0,0,0,0,
↦ 0,0,-1,0, 0,1,0,0,0,0,0, 0,0,0,0,0,0,
↦ 0,0,0, -1,0,0,0,0,0,1,0, 0,0,0,0,0,0,
↦ 0,0,0, 0, 1,0,0,1,0,0,-1,0,0,1,0,0,0
↦ // matrix, 16x4
↦ 0, 0, 1, 0,
↦ 0, y, 0, 0,
↦ 1, 0, 0, 0,
↦ 0, 0, 0, y,
↦ 0, -1,0, 0,
↦ 0, 0, x2,0,
↦ 0, 0, 0, -x2,
↦ 1, 0, 0, 0,
↦ 0, 0, 0, -1,
↦ -1,0, 0, 0,
↦ 0, 1, 0, 0,
↦ 0, 0, 1, 0,
↦ -1,0, 0, 0,
↦ 0, 0, 0, x2y,
↦ 0, 0, x2,0,
↦ 0, -y,0, 0
printlevel = p;
```

D.4.3.3 Ext_R

Procedure from library `homolog.lib` (see Section D.4.3 [homolog_lib], page 375).

Usage: Ext_R(v,M[,p]); v int resp. intvec , M module, p int

Compute:

A presentation of Ext^k(M',R); for k=v[1],v[2],..., M'=coker(M).
Let ...–> F2 –> F1 –M-> F0–>M'–>0
be a free resolution of M'. If
0 <– F0* <-A1– F1* <-A2– F2* <-A3–...
denotes the dual sequence, with Fi*=Hom(Fi,R),
then Ext^k = ker(Ak)/im(Ak+1) is presented as in the following exact sequences:
Fk-1* <-Ak– Fk* <-syz(Ak)– R^p
Fk*/im(Ak+1) <-syz(Ak)– R^p <-Ext^k– R^q

Hence Ext^k=modulo(syz(Ak),Ak+1) presents Ext^k(M',R).

Return:

Ext^k, of type module, a presentation of Ext^k(M',R) if v is of type int, resp. a list of Ext^k (k=v[1],v[2],...) if v is of type intvec.
In case of a third argument of type int return a list l:
  l[1] = module Ext^k resp. list of Ext^k
  l[2] = SB of Ext^k resp. list of SB of Ext^k
  l[3] = matrix resp.list of matrices,
        each representing kbase of Ext^k (if finite dimensional)

Display: printlevel >=0: degree of Ext^k for each k (default) printlevel >=1: Ak, Ak+1 and kbase of Ext^k in Fk*

Note: In order to compute Ext^k(M,R) use the command Ext_R(k,syz(M)); or the 2 commands: list L=mres(M,2); Ext_R(k,L[2]);

Example:

```
LIB "homolog.lib";
int p      = printlevel;
printlevel = 1;
ring r     = 0,(x,y,z),dp;
ideal i    = x2y,y2z,z3x;
module E   = Ext_R(1,i);    //computes Ext^1(r/i,r)
↦ // Computing Ext^1:
↦ // Let 0<--coker(M)<--F0<--F1<--F2<--... be a resolution of M,
↦ // then F1*-->F2* is given by:
↦ x2, -yz,0,
↦ 0,  z3, -xy,
↦ xz2,0,  -y2
↦ // and F0*-->F1* is given by:
↦ y2z,
↦ x2y,
↦ xz3
↦
↦ // degree of Ext^1:
↦ 0
↦
is_zero(E);
↦ 1
module m   = [x],[0,y];
list L     = Ext_R(2..3,m); //computes Ext^i(r^2/m,r), i=2,3
↦ // Computing Ext^2:
↦ // Let 0<--coker(M)<--F0<--F1<--F2<--... be a resolution of M,
↦ // then F2*-->F3* is given by:
↦ 1
↦ // and F1*-->F2* is given by:
↦ 0,0
↦
↦ // degree of Ext^2:
↦ 0
↦
↦ // Computing Ext^3:
↦ // Let 0<--coker(M)<--F0<--F1<--F2<--... be a resolution of M,
↦ // then F3*-->F4* is given by:
```

```
↦ 0
↦ // and F2*-->F3* is given by:
↦ 1
↦
↦ // degree of Ext^3:
↦ 0
↦
show(L);"";
↦ // list, 2 element(s):
↦ [1]:
↦    // module, 1 generator(s)
↦ [1]
↦ [2]:
↦    // module, 1 generator(s)
↦ [1]
↦
qring R    = std(x2+yz);
intvec v   = 0,2,4;
printlevel = 2;             //shows what is going on
ideal i    = x,y,z;         //computes Ext^i(r/(x,y,z),r/(x2+yz)), i=0,2,4
list L     = Ext_R(v,i,1);  //over the qring R=r/(x2+yz), std and kbase
↦ // Computing Ext^0:
↦ // Let 0<--coker(M)<--F0<--F1<--F2<--... be a resolution of M,
↦ // then F0*-->F1* is given by:
↦ z,
↦ y,
↦ x
↦ // and F-1*-->F0* is given by:
↦ 0
↦
↦ // degree of Ext^0:
↦ 0
↦
↦ // columns of matrix are kbase of Ext^0 in F0*:
↦ 0
↦
↦ // Computing Ext^2:
↦ // Let 0<--coker(M)<--F0<--F1<--F2<--... be a resolution of M,
↦ // then F2*-->F3* is given by:
↦ x,-y,z, 0,
↦ z,x, 0, z,
↦ 0,0, x, y,
↦ 0,0, -z,x
↦ // and F1*-->F2* is given by:
↦ y,-z,0,
↦ x,0, -z,
↦ 0,x, -y,
↦ 0,z, x
↦
↦ // degree of Ext^2:
↦ 0
↦
↦ // columns of matrix are kbase of Ext^2 in F2*:
```

```
↦ x,
↦ -z,
↦ 0,
↦ 0
↦
↦ // Computing Ext^4:
↦ // Let 0<--coker(M)<--F0<--F1<--F2<--... be a resolution of M,
↦ // then F4*-->F5* is given by:
↦ x,-y,z, 0,
↦ z,x, 0, z,
↦ 0,0, x, y,
↦ 0,0, -z,x
↦ // and F3*-->F4* is given by:
↦ x, y,-z,0,
↦ -z,x,0, -z,
↦ 0, 0,x, -y,
↦ 0, 0,z, x
↦
↦ // degree of Ext^4:
↦ 0
↦
↦ // columns of matrix are kbase of Ext^4 in F4*:
↦ 0,
↦ 0,
↦ 0,
↦ 0
↦
printlevel = p;
```

D.4.3.4 Ext

Procedure from library `homolog.lib` (see Section D.4.3 [homolog_lib], page 375).

Usage: Ext(v,M,N[,any]); v int resp. intvec, M,N modules

Compute:

```
A presentation of Ext^k(M',N'); for k=v[1],v[2],... where
M'=coker(M) and N'=coker(N). Let
   0<--M'<-- F0 <-M-- F1 <-- F2 <--...  resp.
   0<--N'<-- G0 <--N- G1                be
a free resolution of M' resp. a presentations of N'. Consider

           0                   0                   0
           |^                  |^                  |^
   --> Hom(Fk-1,N') -Ak-> Hom(Fk,N') -Ak+1-> Hom(Fk+1,N')
           |^                  |^                  |^
   --> Hom(Fk-1,G0) -Ak-> Hom(Fk,G0) -Ak+1-> Hom(Fk+1,G0)
                               |^                  |^
                               |C                  |B
                          Hom(Fk,G1) ------> Hom(Fk+1,G1)
(Ak,Ak+1 induced by M and B,C induced by N).

Let K=modulo(Ak+1,B), J=module(Ak)+module(C) and Ext=modulo(K,J),
then we have exact sequences
```

R^p –K-> Hom(Fk,G0) –Ak+1-> Hom(Fk+1,G0)/im(B)
R^q -Ext-> R^p –K->Hom(Fk,G0)/im(Ak)+im(C) –Ak+1->Hom(Fk+1,G0)/im(B)

Hence Ext presents Ext^k(M',N').

Return:

- module Ext, a presentation of Ext^k(M',N') if v is of type int
- a list of Ext^k (k=v[1],v[2],...) if v is of type intvec.
- In case of a third argument of any type return a list l:
  l[1] = module Ext/list of Ext^k
  l[2] = SB of Ext/list of SB of Ext^k
  l[3] = matrix/list of matrices, each representing a kbase of Ext^k
  (if finite dimensional)

Display: printlevel >=0: degree of Ext^k for each k (default).
printlevel >=1: matrices Ak, Ak+1 and kbase of Ext^k in Hom(Fk,G0) (if finite dimensional)

Note: In order to compute Ext^k(M,N) use the command Ext(k,syz(M),syz(N)); or: list P=mres(M,2); list Q=mres(N,2); Ext(k,P[2],Q[2]);

Example:

```
LIB "homolog.lib";
int p      = printlevel;
printlevel = 1;
ring r     = 0,(x,y),dp;
ideal i    = x2-y3;
ideal j    = x2-y5;
list E     = Ext(0..2,i,j);    // Ext^k(r/i,r/j) for k=0,1,2 over r
↦ // Computing Ext^0 (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(F0,G0)-->Hom(F1,G0) is given by:
↦ y3-x2
↦ // and Hom(F-1,G0) + Hom(F0,G1)-->Hom(F0,G0) is given by:
↦ 0,-y5+x2
↦ 
↦ // degree of Ext^0:
↦ 0
↦ 
↦ // Computing Ext^1 (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(F1,G0)-->Hom(F2,G0) is given by:
↦ 0
↦ // and Hom(F0,G0) + Hom(F1,G1)-->Hom(F1,G0) is given by:
↦ y3-x2,-y5+x2
↦ 
↦ // degree of Ext^1:
↦ 0
↦ 
↦ // Computing Ext^2 (help Ext; gives an explanation):
↦ // Let 0<--coker(M)<--F0<--F1<--F2<--... be a resolution of coker(M),
↦ // and 0<--coker(N)<--G0<--G1 a presentation of coker(N),
```

```
↦ // then Hom(F2,G0)-->Hom(F3,G0) is given by:
↦ 1
↦ // and Hom(F1,G0) + Hom(F2,G1)-->Hom(F2,G0) is given by:
↦ 0,-y5+x2
↦
↦ // degree of Ext^2:
↦ 0
↦
qring R    = std(i);
ideal j    = fetch(r,j);
module M   = [-x,y],[-y2,x];
printlevel = 2;
module E1  = Ext(1,M,j);       // Ext^1(R^2/M,R/j) over R=r/i
↦ // Computing Ext^1 (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(F1,G0)-->Hom(F2,G0) is given by:
↦ x, -y,
↦ y2,-x
↦ // and Hom(F0,G0) + Hom(F1,G1)-->Hom(F1,G0) is given by:
↦ x, -y,-y5+x2,0,
↦ y2,-x,0,     -y5+x2
↦
↦ // degree of Ext^1:
↦ 0
↦
list l     = Ext(4,M,M,1);     // Ext^4(R^2/M,R^2/M) over R=r/i
↦ // Computing Ext^4 (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(F4,G0)-->Hom(F5,G0) is given by:
↦ x, -y,0, 0,
↦ y2,-x,0, 0,
↦ 0, 0, x, -y,
↦ 0, 0, y2,-x
↦ // and Hom(F3,G0) + Hom(F4,G1)-->Hom(F4,G0) is given by:
↦ x, -y,0, 0, -x,0, -y2,0,
↦ y2,-x,0, 0, 0, -x,0,  -y2,
↦ 0, 0, x, -y,y, 0, x,  0,
↦ 0, 0, y2,-x,0, y, 0,  x
↦
↦ // degree of Ext^4:
↦ 0
↦
↦ // columns of matrix are kbase of Ext^4 in Hom(F4,G0)
↦ 1,0,
↦ 0,y,
↦ 0,1,
↦ 1,0
↦
↦ // element 1 of kbase of Ext^4 in Hom(F4,G0)
↦ // as matrix: F4-->G0
↦ 1,0,
```

```
↦ 0,1
↦ // element 2 of kbase of Ext^4 in Hom(F4,G0)
↦ // as matrix: F4-->G0
↦ 0,y,
↦ 1,0
↦
printlevel = p;
```

D.4.3.5 Hom

Procedure from library `homolog.lib` (see Section D.4.3 [homolog_lib], page 375).

Usage: Hom(M,N,[any]); M,N=modules

Compute:

A presentation of Hom(M',N'), M'=coker(M), N'=coker(N) as follows:
Let ...–>F1 –M-> F0–>M'–>0 and ...–>G1 –N-> G0–>N'–>0 be presentations of M' and N'. Consider

```
                              0               0
                              |^              |^
   0 -> Hom(M',N') ---> Hom(F0,N') ---> Hom(F1,N')
                              |^              |^
(A: induced by M)        Hom(F0,G0) -A-> Hom(F1,G0)
                              |^              |^
(B,C:induced by N)            |C              |B
                         Hom(F0,G1) ---> Hom(F1,G1)
```

Let D=modulo(A,B) and Hom=modulo(D,C), then we have exact sequences

R^p –D-> Hom(F0,G0) –A-> Hom(F1,G0)/im(B)
R^q -Hom-> R^p –D-> Hom(F0,G0)/im(C) –A-> Hom(F1,G0)/im(B).

Hence Hom presents Hom(M',N')

Return: module Hom, a presentation of Hom(M',N'), resp., in case of 3 arguments, a list l:
- l[1] = Hom
- l[2] = SB of Hom
- l[3] = kbase of coker(Hom) (if finite dimensional), represented by elements in Hom(F0,G0) via mapping D

Display: printlevel >=0: degree of Hom (default)
printlevel >=1: D and C and kbase of coker(Hom) in Hom(F0,G0)
printlevel >=2: elements of kbase of coker(Hom) as matrix :F0–>G0

Note: DISPLAY is as described only for a direct call of 'Hom'. Calling 'Hom' from another proc has the same effect as decreasing printlevel by 1.

Example:

```
LIB "homolog.lib";
int p     = printlevel;
printlevel= 1;   //in 'example proc' printlevel has to be increased by 1
ring r    = 0,(x,y),dp;
ideal i   = x2-y3,xy;
qring q   = std(i);
```

```
ideal i   = fetch(r,i);
module M  = [-x,y],[-y2,x],[x3];
module H  = Hom(M,i);
↦ // degree of Hom:
↦ 0
↦
↦ // given ...--> F1 --M-> F0 -->M'--> 0 and ...--> G1 --N-> G0 -->N'--> 0,
↦ // show D=ker(Hom(F0,G0) --> Hom(F1,G0)/im(Hom(F1,G1) --> Hom(F1,G0)))
↦ y,x, 0,
↦ x,y2,x2
↦ // show C=im(Hom(F0,G1) --> Hom(F0,G0))
↦ -y3+x2,0,     xy,0,
↦ 0,     -y3+x2,0, xy
↦
print(H);
↦ 0, x, 0,y2,0,
↦ y, 0, 0,-x,x2,
↦ -1,-1,x,0, 0
printlevel= 2;
list L    = Hom(M,i,1);"";
↦ // degree of Hom:
↦ 0
↦
↦ // given ...--> F1 --M-> F0 -->M'--> 0 and ...--> G1 --N-> G0 -->N'--> 0,
↦ // show D=ker(Hom(F0,G0) --> Hom(F1,G0)/im(Hom(F1,G1) --> Hom(F1,G0)))
↦ y,x, 0,
↦ x,y2,x2
↦ // show C=im(Hom(F0,G1) --> Hom(F0,G0))
↦ -y3+x2,0,     xy,0,
↦ 0,     -y3+x2,0, xy
↦
↦ // element 1 of kbase of Hom in Hom(F0,G0) as matrix: F0-->G0:
↦ y2,xy
↦ // element 2 of kbase of Hom in Hom(F0,G0) as matrix: F0-->G0:
↦ y,x
↦ // element 3 of kbase of Hom in Hom(F0,G0) as matrix: F0-->G0:
↦ x2,xy2
↦ // element 4 of kbase of Hom in Hom(F0,G0) as matrix: F0-->G0:
↦ x,y2
↦ // element 5 of kbase of Hom in Hom(F0,G0) as matrix: F0-->G0:
↦ 0,x2
↦
ring s    = 3,(x,y,z),(c,dp);
ideal i   = jacob(ideal(x2+y5+z4));
qring rq=std(i);
matrix M[2][2]=xy,x3,5y,4z,x2;
matrix N[3][2]=x2,x,y3,3xz,x2z,z;
print(M);
↦ xy,x3,
↦ -y,z
print(N);
↦ x2, x,
↦ y3, 0,
```

```
↦ x2z,z
list l=Hom(M,N,1);
↦ // degree of Hom:
↦ 0
↦
↦ // given ...--> F1 --M-> F0 -->M'--> 0 and ...--> G1 --N-> G0 -->N'--> 0,
↦ // show D=ker(Hom(F0,G0) --> Hom(F1,G0)/im(Hom(F1,G1) --> Hom(F1,G0)))
↦ 0,0, 0,0, 0,   0,0,   1,
↦ 0,0, 0,0, 0,   0,y3z2,0,
↦ 0,0, 0,0, 0,   1,0,   0,
↦ 0,0, 0,y3,y2z2,0,0,   0,
↦ 0,0, 1,0, 0,   0,0,   0,
↦ z,y3,0,0, 0,   0,0,   0
↦ // show C=im(Hom(F0,G1) --> Hom(F0,G0))
↦ x2, 0,  x,0,
↦ 0,  x2, 0,x,
↦ y3, 0,  0,0,
↦ 0,  y3, 0,0,
↦ x2z,0,  z,0,
↦ 0,  x2z,0,z
↦
↦ // element 1 of kbase of Hom in Hom(F0,G0) as matrix: F0-->G0:
↦ 0,0,
↦ 0,0,
↦ 0,y3
↦ // element 2 of kbase of Hom in Hom(F0,G0) as matrix: F0-->G0:
↦ 0, 0,
↦ 0, 0,
↦ y3,0
↦ // element 3 of kbase of Hom in Hom(F0,G0) as matrix: F0-->G0:
↦ 0, 0,
↦ 0, 0,
↦ y2,0
↦ // element 4 of kbase of Hom in Hom(F0,G0) as matrix: F0-->G0:
↦ 0,0,
↦ 0,0,
↦ y,0
↦ // element 5 of kbase of Hom in Hom(F0,G0) as matrix: F0-->G0:
↦ 0,0,
↦ 0,0,
↦ 1,0
↦ // element 6 of kbase of Hom in Hom(F0,G0) as matrix: F0-->G0:
↦ 0,0,
↦ 0,y2z2,
↦ 0,0
↦ // element 7 of kbase of Hom in Hom(F0,G0) as matrix: F0-->G0:
↦ 0,   0,
↦ y2z2,0,
↦ 0,   0
↦ // element 8 of kbase of Hom in Hom(F0,G0) as matrix: F0-->G0:
↦ 0,  0,
↦ yz2,0,
↦ 0,  0
```

```
↦ // element 9 of kbase of Hom in Hom(F0,G0) as matrix: F0-->G0:
↦ 0, 0,
↦ z2,0,
↦ 0, 0
↦ // element 10 of kbase of Hom in Hom(F0,G0) as matrix: F0-->G0:
↦ 0,  0,
↦ y2z,0,
↦ 0,  0
↦ // element 11 of kbase of Hom in Hom(F0,G0) as matrix: F0-->G0:
↦ 0, 0,
↦ yz,0,
↦ 0, 0
↦ // element 12 of kbase of Hom in Hom(F0,G0) as matrix: F0-->G0:
↦ 0,0,
↦ z,0,
↦ 0,0
↦ // element 13 of kbase of Hom in Hom(F0,G0) as matrix: F0-->G0:
↦ 0, 0,
↦ y2,0,
↦ 0, 0
↦ // element 14 of kbase of Hom in Hom(F0,G0) as matrix: F0-->G0:
↦ 0,0,
↦ y,0,
↦ 0,0
↦ // element 15 of kbase of Hom in Hom(F0,G0) as matrix: F0-->G0:
↦ 0,0,
↦ 1,0,
↦ 0,0
↦ // element 16 of kbase of Hom in Hom(F0,G0) as matrix: F0-->G0:
↦ 0,y3z2,
↦ 0,0,
↦ 0,0
printlevel = p;
```

D.4.3.6 homology

Procedure from library `homolog.lib` (see Section D.4.3 [homolog_lib], page 375).

Usage: homology(A,B,M,N);

Compute: Let M and N be submodules of R^m and R^n presenting M'=R^m/M, N'=R^n/N (R=basering) and let A,B matrices inducing maps R^k–A->R^m–B->R^n. Compute a presentation R^q –H-> R^m of the module
ker(B)/im(A) := ker(M'/im(A) –B-> N'/im(BM)+im(BA)).
If B induces a map M'–B->N' (i.e BM=0) and if R^k–A->M'–B->N' is a complex (i.e. BA=0) then ker(B)/im(A) is the homology of the complex

Return: module H, a presentation of ker(B)/im(A)

Note: homology returns a free module of rank m if ker(B)=im(A)

Example:

```
LIB "homolog.lib";
ring r;
ideal id=maxideal(4);
```

```
qring qr=std(id);
module N=maxideal(3)*freemodule(2);
module M=maxideal(2)*freemodule(2);
module B=[2x,0],[x,y],[z2,y];
module A=M;
degree(std(homology(A,B,M,N)));"";
↦ 0
↦
ring s=0,x,ds;
qring qs=std(x4);
module A=[x];module B=A;
module M=[x3];module N=M;
homology(A,B,M,N);
↦ _[1]=gen(1)
```

D.4.3.7 kernel

Procedure from library `homolog.lib` (see Section D.4.3 [homolog_lib], page 375).

Usage: kernel(A,M,N);

Compute: Let M and N be submodules of R^m and R^n presenting M'=R^m/M, N'=R^n/N (R=basering) and let A:R^m–>R^n a matrix inducing a map A':M'–>N'. Compute a presentation K of ker(A') as in the commutative diagram:

```
ker(A') --->  M' --A'--> N'
   |^         |^         |^
   |          |          |
  R^r  ---> R^m --A--> R^n
   |^         |^         |^
   |K         |M         |N
   |          |          |
  R^s  ---> R^p -----> R^q
```

Return: module K, a presentation of ker(A')

Example:

```
LIB "homolog.lib";
ring r;
module N=[2x,x],[0,y];
module M=maxideal(1)*freemodule(2);
matrix A[2][2]=2x,0,x,y,z2,y;
module K=kernel(A,M,N);
degree(std(K));
↦ 0
print(K);
↦ z,0,y,0,x,0,
↦ 0,z,0,y,0,x
```

D.4.3.8 kohom

Procedure from library `homolog.lib` (see Section D.4.3 [homolog_lib], page 375).

Usage: kohom(A,k); A=matrix, k=integer

Return: matrix Hom(R^k,A), i.e. let A be a matrix defining a map: F1 –> F2 of free R-modules, then matrix of Hom(R^k,F1)–>Hom(R^k,F2) is computed.

Example:

```
LIB "homolog.lib";
ring r;
matrix n[2][3]=x,y,5,z,77,33;
print(kohom(n,3));
↦ x,0,0,y, 0, 0, 5, 0, 0,
↦ 0,x,0,0, y, 0, 0, 5, 0,
↦ 0,0,x,0, 0, y, 0, 0, 5,
↦ z,0,0,77,0, 0, 33,0, 0,
↦ 0,z,0,0, 77,0, 0, 33,0,
↦ 0,0,z,0, 0, 77,0, 0, 33
```

D.4.3.9 kontrahom

Procedure from library `homolog.lib` (see Section D.4.3 [homolog_lib], page 375).

Usage: kontrahom(A,k); A=matrix, k=integer

Return: matrix Hom(A,R^k), i.e. let A be a matrix defining a map: F1 –> F2 of free R-modules, the matrix of Hom(F2,R^k)–>Hom(F1,R^k) is computed

Example:

```
LIB "homolog.lib";
ring r;
matrix n[2][3]=x,y,5,z,77,33;
print(kontrahom(n,3));
↦ x,z, 0,0, 0,0,
↦ y,77,0,0, 0,0,
↦ 5,33,0,0, 0,0,
↦ 0,0, x,z, 0,0,
↦ 0,0, y,77,0,0,
↦ 0,0, 5,33,0,0,
↦ 0,0, 0,0, x,z,
↦ 0,0, 0,0, y,77,
↦ 0,0, 0,0, 5,33
```

D.4.4 mprimdec_lib

Library: mprimdec.lib

Purpose: procedures for primary decomposition of modules

Authors: Alexander Dreyer, dreyer@mathematik.uni-kl.de; adreyer@web.de

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

Procedures:

D.4.4.1 separator

Procedure from library `mprimdec.lib` (see Section D.4.4 [mprimdec_lib], page 391).

Usage: separator(l); list l of prime ideals

Return: list sepList;
a list of separators of the prime ideals in l,
i.e. polynomials p_ij, s.th. p_ij is in l[j],
for all l[j] not contained in l[i]
but p_ij is not in l[i]

Example:

```
LIB "mprimdec.lib";
ring r=0,(x,y,z),dp;
ideal i=(x2y,xz2,y2z,z3);
list l=minAssGTZ(i);
list sepL=separator(l);
sepL;
↦ [1]:
↦    x
↦ [2]:
↦    y
```

D.4.4.2 PrimdecA

Procedure from library `mprimdec.lib` (see Section D.4.4 [mprimdec_lib], page 391).

Usage: PrimdecA (N[, i]); module N, int i

Return: list l
a (not necessarily minimal) primary decomposition of N computed by a generalized version of
the algorithm of Schimoyama/Yokoyama,
if i=1 is given, the factorizing Groebner is used
to compute the isolated primes.

Example:

```
LIB "mprimdec.lib";
ring r=0,(x,y,z),dp;
module N=x*gen(1)+ y*gen(2),
x*gen(1)-x2*gen(2);
list l=PrimdecA(N);
l;
↦ [1]:
↦    [1]:
↦       _[1]=x*gen(1)+y*gen(2)
↦       _[2]=x*gen(2)-gen(1)
↦    [2]:
↦       _[1]=x2+y
↦ [2]:
↦    [1]:
↦       _[1]=gen(2)
↦       _[2]=x*gen(1)
↦    [2]:
↦       _[1]=x
↦ [3]:
↦    [1]:
↦       _[1]=y*gen(1)
↦       _[2]=y*gen(2)
```

```
↦        _[3]=x*gen(1)
↦        _[4]=x*gen(2)
↦     [2]:
↦        _[1]=y
↦        _[2]=x
```

D.4.4.3 PrimdecB

Procedure from library `mprimdec.lib` (see Section D.4.4 [mprimdec_lib], page 391).

Usage: PrimdecB (N, p); pseudo-primary module N, isolated prime ideal p

Return: list l
a (not necessarily minimal) primary decomposition of N

Example:

```
LIB "mprimdec.lib";
ring r=0,(x,y,z),dp;
module N=y*gen(1),y2*gen(2),yz*gen(2),yx*gen(2);
ideal p=y;
list l=PrimdecB(N,p);
l;
↦ [1]:
↦    [1]:
↦       _[1]=y*gen(1)
↦       _[2]=y*gen(2)
↦    [2]:
↦       _[1]=y
↦ [2]:
↦    [1]:
↦       _[1]=y*gen(1)
↦       _[2]=y*gen(2)
↦       _[3]=x*gen(1)
↦       _[4]=x*gen(2)
↦    [2]:
↦       _[1]=y
↦       _[2]=x
↦ [3]:
↦    [1]:
↦       _[1]=z*gen(1)
↦       _[2]=z*gen(2)
↦       _[3]=y*gen(1)
↦       _[4]=x*gen(1)
↦       _[5]=x*gen(2)
↦       _[6]=y2*gen(2)
↦    [2]:
↦       _[1]=x
↦       _[2]=y
↦       _[3]=z
```

D.4.4.4 modDec

Procedure from library `mprimdec.lib` (see Section D.4.4 [mprimdec_lib], page 391).

Usage: modDec (N[, i]); module N, int i

Return: list l
a minimal primary decomposition of N
computed by an generalized version of
the algorithm of Schimoyama/Yokoyama,
if i=1 is given, the factorizing Groebner is used

Example:

```
LIB "mprimdec.lib";
ring r=0,(x,y,z),dp;
module N=x*gen(1)+ y*gen(2),
x*gen(1)-x2*gen(2);
list l=modDec(N);
l;
↦ [1]:
↦    [1]:
↦       _[1]=x*gen(1)+y*gen(2)
↦       _[2]=x*gen(2)-gen(1)
↦    [2]:
↦       _[1]=x2+y
↦ [2]:
↦    [1]:
↦       _[1]=gen(2)
↦       _[2]=x*gen(1)
↦    [2]:
↦       _[1]=x
```

D.4.4.5 zeroMod

Procedure from library `mprimdec.lib` (see Section D.4.4 [mprimdec_lib], page 391).

Usage: zeroMod (N[, check]); zero-dimensional module N[, module check]

Return: list l
the minimal primary decomposition of a zero-dimensional module N, computed by a generalized version of the algorithm
of Gianni, Trager and Zacharias

Note: if the parameter check is given, only components
not containing check are computed

Example:

```
LIB "mprimdec.lib";
ring r=0,z,dp;
module N=z*gen(1),(z-1)*gen(2),(z+1)*gen(3);
list l=zeroMod(N);
↦ 2
l;
↦ [1]:
↦    [1]:
↦       _[1]=gen(2)
↦       _[2]=gen(3)
↦       _[3]=z*gen(1)
↦    [2]:
↦       _[1]=z
↦ [2]:
```

```
↦    [1]:
↦       _[1]=gen(1)
↦       _[2]=gen(2)
↦       _[3]=z*gen(3)+gen(3)
↦    [2]:
↦       _[1]=z+1
↦ [3]:
↦    [1]:
↦       _[1]=gen(1)
↦       _[2]=gen(3)
↦       _[3]=z*gen(2)-gen(2)
↦    [2]:
↦       _[1]=z-1
```

D.4.4.6 GTZmod

Procedure from library `mprimdec.lib` (see Section D.4.4 [mprimdec_lib], page 391).

Usage: GTZmod (N[, check]); module N[, module check]

Return: list l
the minimal primary decomposition of the module N,
computed by a generalized version of the algorithm
of Gianny, Trager and Zacharias

Note: if the parameter check is given, only components
not containing check are computed

Example:

```
LIB "mprimdec.lib";
ring r=0,(x,y,z),dp;
module N=x*gen(1)+ y*gen(2),
x*gen(1)-x2*gen(2);
list l=GTZmod(N);
↦ 2
l;
↦ [1]:
↦    [1]:
↦       _[1]=x*gen(1)+y*gen(2)
↦       _[2]=x*gen(2)-gen(1)
↦    [2]:
↦       _[1]=x2+y
↦ [2]:
↦    [1]:
↦       _[1]=gen(2)
↦       _[2]=x*gen(1)
↦    [2]:
↦       _[1]=x
```

D.4.4.7 dec1var

Procedure from library `mprimdec.lib` (see Section D.4.4 [mprimdec_lib], page 391).

Usage: dec1var (N); zero-dimensional module N[, module check]

Return: list l
the minimal primary decomposition of a submodule N of R^s if nvars(R)=1

Note: if the parameter check is given, only components
not containing check are computed

Example:

```
LIB "mprimdec.lib";
ring r=0,z,dp;
module N=z*gen(1),(z-1)*gen(2),(z+1)*gen(3);
list l=dec1var(N);
l;
↦ [1]:
↦    [1]:
↦       _[1]=gen(2)
↦       _[2]=gen(3)
↦       _[3]=z*gen(1)
↦    [2]:
↦       _[1]=z
↦ [2]:
↦    [1]:
↦       _[1]=gen(1)
↦       _[2]=gen(2)
↦       _[3]=z*gen(3)+gen(3)
↦    [2]:
↦       _[1]=z+1
↦ [3]:
↦    [1]:
↦       _[1]=gen(1)
↦       _[2]=gen(3)
↦       _[3]=z*gen(2)-gen(2)
↦    [2]:
↦       _[1]=z-1
```

D.4.4.8 annil

Procedure from library `mprimdec.lib` (see Section D.4.4 [mprimdec_lib], page 391).

Usage: annil(N); module N

Return: ideal ann=std(quotient(N,freemodule(nrows(N))));
the annihilator of M/N in the basering

Note: ann is a std basis in the basering

Example:

```
LIB "mprimdec.lib";
ring r=0,(x,y,z),dp;
module N=x*gen(1), y*gen(2);
ideal ann=annil(N);
ann;
↦ ann[1]=xy
```

D.4.4.9 splitting

Procedure from library `mprimdec.lib` (see Section D.4.4 [mprimdec_lib], page 391).

Usage: splitting(N[,check[, ann]]); module N, module check, ideal ann

Return: (l, check) list l, module check
the elements of l consists of a triple with
[1] of type module [2] and [3] of type ideal
s.th. the intersection of the modules is equal to the zero-dimensional module N, furthermore l[j][3]=annil(l[j][1]) if l[j][2]!=0 then the module l[j][1] is primary
with associated prime l[j][2],
and check=intersect(check, l[j][1]) is computed

Note: if the parameter check is given, only components not containing check are computed; if ann is given, ann is used instead of annil(N)

Example:

```
LIB "mprimdec.lib";
ring r=0,z,lp;
module N=z*gen(1), (z+1)*gen(2);
N=std(N);
list l; module check;
(l, check)=splitting(N);
l;
↦ [1]:
↦    [1]:
↦       _[1]=gen(2)
↦       _[2]=z*gen(1)
↦    [2]:
↦       _[1]=z
↦    [3]:
↦       _[1]=z
↦    [4]:
↦       _[1]=z
↦ [2]:
↦    [1]:
↦       _[1]=gen(1)
↦       _[2]=z*gen(2)+gen(2)
↦    [2]:
↦       _[1]=z+1
↦    [3]:
↦       _[1]=z+1
↦    [4]:
↦       _[1]=z+1
check;
↦ check[1]=z*gen(2)+gen(2)
↦ check[2]=z*gen(1)
```

D.4.4.10 primTest

Procedure from library `mprimdec.lib` (see Section D.4.4 [mprimdec_lib], page 391).

Usage: primTest(i[, p]); a zero-dimensional ideal i, irreducible poly p in i

Return: if i neither is prime nor is homogeneous then ideal(0) is returned, else radical(i)

Example:

```
LIB "mprimdec.lib";
```

```
ring r=0,(x,y,z),lp;
ideal i=x+1,y-1,z;
i=std(i);
ideal primId=primTest(i,z);
primId;
↦ primId[1]=z
↦ primId[2]=y-1
↦ primId[3]=x+1
i=x,z2,yz,y2;
i=std(i);
primId=primTest(i);
primId;
↦ primId[1]=x
↦ primId[2]=y
↦ primId[3]=z
```

D.4.4.11 preComp

Procedure from library `mprimdec.lib` (see Section D.4.4 [mprimdec_lib], page 391).

Usage: preComp(N,check[, ann]); module N, module check, ideal ann

Return: (l, check) list l, module check
the elements of l consists of a triple with
[1] of type module [2] and [3] of type ideal
s.th. the intersection of the modules is equal to the zero-dimensional module N, furthermore l[j][3]=annil(l[j][1]) if l[j][2]!=0 then the module l[j][1] is primary
with associated prime l[j][2],
and check=intersect(check, l[j][1]) is computed

Note: only components not containing check are computed;
if ann is given, ann is used instead of annil(N)

Example:

```
LIB "mprimdec.lib";
ring r=0,z,lp;
module N=z*gen(1), (z+1)*gen(2);
N=std(N);
list l; module check;
(l, check)=preComp(N,freemodule(2));
l;
↦ [1]:
↦    [1]:
↦       _[1]=z*gen(1)
↦       _[2]=gen(2)
↦    [2]:
↦       _[1]=z
↦    [3]:
↦       _[1]=z
↦ [2]:
↦    [1]:
↦       _[1]=gen(1)
↦       _[2]=z*gen(2)+gen(2)
↦    [2]:
```

```
↦        _[1]=z+1
↦     [3]:
↦        _[1]=z+1
check;
↦ check[1]=z*gen(1)
↦ check[2]=z*gen(2)+gen(2)
```

D.4.4.12 indSet

Procedure from library `mprimdec.lib` (see Section D.4.4 [mprimdec_lib], page 391).

Usage: indSet(i); i ideal

Return: list with two entrees
both are lists of new varstrings with the dependent variables the independent set, the ordstring with the corresp. block ordering, and the integer where the independent set starts in the varstring

Note: the first entry gives the strings for all maximal independent sets the second gives the strings for the independent sets,
which cannot be enhanced

Example:

```
LIB "mprimdec.lib";
ring s1=(0,x,y),(a,b,c,d,e,f,g),lp;
ideal i=ea-fbg,fa+be,ec-fdg,fc+de;
i=std(i);
list  l=indSet(i);
l;
↦ [1]:
↦    [1]:
↦       [1]:
↦          e,f
↦       [2]:
↦          a,b,c,d,g
↦       [3]:
↦          (C,dp(2),dp)
↦       [4]:
↦          5
↦ [2]:
↦    [1]:
↦       [1]:
↦          a,b,c,d
↦       [2]:
↦          e,f,g
↦       [3]:
↦          (C,dp(4),dp)
↦       [4]:
↦          3
↦    [2]:
↦       [1]:
↦          a,c,e
↦       [2]:
↦          b,d,f,g
↦       [3]:
```

```
↦          (C,dp(3),dp)
↦       [4]:
↦          4
```

D.4.4.13 GTZopt

Procedure from library `mprimdec.lib` (see Section D.4.4 [mprimdec_lib], page 391).

Usage: GTZopt (N[, check]); module N[, module check]

Return: list l
the minimal primary decomposition of the module N,
computed by a generalized and optimized version of
the algorithm of Gianny, Trager and Zacharias

Note: if the parameter check is given, only components
not containing check are computed

Example:

```
LIB "mprimdec.lib";
ring r=0,(x,y,z),dp;
module N=x*gen(1)+ y*gen(2),
x*gen(1)-x2*gen(2);
list l=GTZopt(N);
l;
↦ [1]:
↦    [1]:
↦       _[1]=x*gen(1)+y*gen(2)
↦       _[2]=x*gen(2)-gen(1)
↦    [2]:
↦       _[1]=x2+y
↦ [2]:
↦    [1]:
↦       _[1]=gen(2)
↦       _[2]=x*gen(1)
↦    [2]:
↦       _[1]=x
```

D.4.4.14 zeroOpt

Procedure from library `mprimdec.lib` (see Section D.4.4 [mprimdec_lib], page 391).

Usage: zeroOpt (N[, check]); zero-dimensional module N[, module check]

Return: list l
the minimal primary decomposition of a zero-dimensional module N, computed by a generalized and optimized version of the algorithm of Gianny, Trager and Zacharias

Note: if the parameter check is given, only components
not containing check are computed

Example:

```
LIB "mprimdec.lib";
ring r=0,z,dp;
module N=z*gen(1),(z-1)*gen(2),(z+1)*gen(3);
list l=zeroOpt(N);
```

```
l;
↦ [1]:
↦    [1]:
↦       _[1]=z*gen(1)
↦       _[2]=gen(2)
↦       _[3]=gen(3)
↦    [2]:
↦       _[1]=z
↦ [2]:
↦    [1]:
↦       _[1]=gen(1)
↦       _[2]=gen(2)
↦       _[3]=z*gen(3)+gen(3)
↦    [2]:
↦       _[1]=z+1
↦ [3]:
↦    [1]:
↦       _[1]=gen(1)
↦       _[2]=z*gen(2)-gen(2)
↦       _[3]=gen(3)
↦    [2]:
↦       _[1]=z-1
```

D.4.4.15 clrSBmod

Procedure from library `mprimdec.lib` (see Section D.4.4 [mprimdec_lib], page 391).

Usage: clrSBmod(N); N module which is SB ordered by monomial ordering

Return: module = minimal SB

Example:

```
LIB "mprimdec.lib";
ring  r = (0,a,b),(x,y,z),dp;
module N1=ax2+y,a2x+y,bx;
module N2=clrSBmod(N1);
N2;
↦ N2[1]=(a)*x2*gen(1)+y*gen(1)
↦ N2[2]=(b)*x*gen(1)
```

D.4.4.16 minSatMod

Procedure from library `mprimdec.lib` (see Section D.4.4 [mprimdec_lib], page 391).

Usage: minSatMod(N, I); module N, ideal I

Return: list with 2 elements:
[1]=sat(N,product(I))[1],
[2]=p, the polynomial of minimal degree s.th. [1]=quotient(N,p)

Example:

```
LIB "mprimdec.lib";
ring  r = 0,(x,y,z),dp;
module N=xy*gen(1);
ideal h=yz,z2;
```

```
list l=minSatMod(N,h);
l;
↦ [1]:
↦    _[1]=x*gen(1)
↦ [2]:
↦    y
```

D.4.4.17 specialModulesEqual

Procedure from library `mprimdec.lib` (see Section D.4.4 [mprimdec_lib], page 391).

Usage: specialModulesEqual(N1, N2) N1, N2 standard bases of modules, s.th. N1 is contained in N2 or vice versa

Return: int i
if (N1==N2) then i=1
else i=0

Example:

```
LIB "mprimdec.lib";
ring  r = 0,(x,y,z),dp;
module N1=x*freemodule(2);
module N2=xy*freemodule(2);
int i=specialModulesEqual(N1,N2);
i;
↦ 0
N2=N1;
i=specialModulesEqual(N1,N2);
i;
↦ 1
```

D.4.4.18 stdModulesEqual

Procedure from library `mprimdec.lib` (see Section D.4.4 [mprimdec_lib], page 391).

Usage: stdModulesEqual(N1, N2) N1, N2 standard bases of modules,

Return: int i
if (N1==N2) then i=1
else i=0

Example:

```
LIB "mprimdec.lib";
ring  r = 0,(x,y,z),dp;
module N1=x*freemodule(2);
module N2=xy*freemodule(2);
int i=stdModulesEqual(N1,N2);
i;
↦ 0
N2=N1;
i=stdModulesEqual(N1,N2);
i;
↦ 1
```

D.4.4.19 modulesEqual

Procedure from library `mprimdec.lib` (see Section D.4.4 [mprimdec_lib], page 391).

Usage: modulesEqual(N1, N2) N1, N2 modules,

Return: int i
if (N1==N2) then i=1
else i=0

Example:

```
LIB "mprimdec.lib";
ring  r = 0,(x,y,z),dp;
module N1=x*freemodule(2);
module N2=xy*freemodule(2);
int i=modulesEqual(N1,N2);
i;
↦ 0
N2=N1;
i=modulesEqual(N1,N2);
i;
↦ 1
```

D.4.4.20 getData

Procedure from library `mprimdec.lib` (see Section D.4.4 [mprimdec_lib], page 391).

Usage: getData(N, l[, noCheck]); module N, list l[, int noCheck]

Return: (ann, check, M, checked)
ideal ann, module check, M, int checked

if l[1] is contained in N [and noCheck is not given]
then checked=1, ann=ideal(0), check=0, M=0;
else checked=0, M=freemodule(nrows(N)); check=l[1]
(resp. check=M if l is an empty list) and
if size(l)>1 then ann=l[2] else ann is the annihilator of M/N.

Note: ann is a std basis in the basering

Example:

```
LIB "mprimdec.lib";
ring  r = 0,(x,y,z),lp;
module N=x*gen(1),y*gen(2);
N=std(N);
ideal ann; module check, M; int checked; list l;
(ann, check, M, checked)=getData(N,l);
ann; check; M; checked;
↦ ann[1]=xy
↦ check[1]=gen(1)
↦ check[2]=gen(2)
↦ M[1]=gen(1)
↦ M[2]=gen(2)
↦ 0
l=list(check,ann);
(ann, check, M, checked)=getData(N,l);
```

```
ann; check; M; checked;
↦ ann[1]=xy
↦ check[1]=gen(1)
↦ check[2]=gen(2)
↦ M[1]=gen(1)
↦ M[2]=gen(2)
↦ 0
l=list(N);
(ann, check, M, checked)=getData(N,l);
ann; check; M; checked;
↦ ann[1]=0
↦ check[1]=0
↦ M[1]=0
↦ 1
```

D.4.5 mregular_lib

Library: mregular.lib

Purpose: Castelnuovo-Mumford Regularity of CM-Schemes and Curves

Authors: I.Bermejo, ibermejo@ull.es
Ph.Gimenez, pgimenez@agt.uva.es
G.-M.Greuel, greuel@mathematik.uni-kl.de

Overview: A library for computing the Castelnuovo-Mumford regularity of a subscheme of the projective n-space that DOES NOT require the computation of a minimal graded free resolution of the saturated ideal defining the subscheme. The procedures are based on two papers by Isabel Bermejo and Philippe Gimenez: 'On Castelnuovo-Mumford regularity of projective curves' Proc.Amer.Math.Soc. 128(5) (2000), and 'Computing the Castelnuovo-Mumford regularity of some subschemes of Pn using quotients of monomial ideals', Proceedings of MEGA-2000, J. Pure Appl. Algebra (to appear).
The algorithm assumes the variables to be in Noether position.

Procedures:

D.4.5.1 reg_CM

Procedure from library `mregular.lib` (see Section D.4.5 [mregular_lib], page 404).

Usage: reg_CM (i); i ideal

Return: an integer, the Castelnuovo-Mumford regularity of i-sat.

Assume: i is a homogeneous ideal of the basering S=K[x(0)..x(n)] where the field K is infinite, and S/i-sat is Cohen-Macaulay. Assume that K[x(n-d),...,x(n)] is a Noether normalization of S/i-sat where d=dim S/i -1. If this is not the case, compute a Noether normalization e.g. by using the proc noetherNormal from algebra.lib.

Note: The output is reg(X)=reg(i-sat) where X is the arithmetically Cohen-Macaulay subscheme of the projective n-space defined by i. If printlevel > 0 (default = 0) additional information is displayed. In particular, the value of the regularity of the Hilbert function of S/i-sat is given.

Example:

```
LIB "mregular.lib";
ring s=0,x(0..5),dp;
ideal i=x(2)^2-x(4)*x(5),x(1)*x(2)-x(0)*x(5),x(0)*x(2)-x(1)*x(4),
x(1)^2-x(3)*x(5),x(0)*x(1)-x(2)*x(3),x(0)^2-x(3)*x(4);
reg_CM(i);
↦ 2
// Additional information can be obtained as follows:
printlevel = 1;
reg_CM(i);
↦ // Ideal i of S defining an arithm. Cohen-Macaulay subscheme X of P5:
↦ //   - dimension of X: 2
↦ //   - i is saturated: YES
↦ //   - regularity of the Hilbert function of S/i-sat: -1
↦ //   - time for computing reg(X): 0 sec.
↦ // Castelnuovo-Mumford regularity of X:
↦ 2
```

D.4.5.2 reg_curve

Procedure from library `mregular.lib` (see Section D.4.5 [mregular_lib], page 404).

Usage: reg_curve (i[,e]); i ideal, e integer

Return: an integer, the Castelnuovo-Mumford regularity of i-sat.

Assume: i is a homogeneous ideal of the basering S=K[x(0)..x(n)] where the field K is infinite, and it defines a projective curve C in the projective n-space (dim(i)=2). We assume that K[x(n-1),x(n)] is a Noether normalization of S/i-sat.
e=0: (default)
Uses a random choice of an element of K when it is necessary. This is absolutely safe (if the element is bad, another random choice will be done until a good element is found).
e=1: Substitutes the random choice of an element of K by a simple transcendental field extension of K.

Note: The output is the integer reg(C)=reg(i-sat).
If printlevel > 0 (default = 0) additional information is displayed. In particular, says if C is arithmetically Cohen-Macaulay or not, determines in which step of a minimal graded free resolution of i-sat the regularity of C is attained, and sometimes gives the value of the regularity of the Hilbert function of S/i-sat (otherwise, an upper bound is given).

Example:

```
LIB "mregular.lib";
ring s = 0,(x,y,z,t),dp;
// 1st example is Ex.2.5 in [Bermejo-Gimenez], Proc.Amer.Math.Soc. 128(5):
ideal i  = x17y14-y31, x20y13, x60-y36z24-x20z20t20;
reg_curve(i);
↦ 72
// 2nd example is Ex.2.9 in [Bermejo-Gimenez], Proc.Amer.Math.Soc. 128(5):
int k=43;
ideal j=x17y14-y31,x20y13,x60-y36z24-x20z20t20,y41*z^k-y40*z^(k+1);
reg_curve(j);
↦ 93
// Additional information can be obtained as follows:
printlevel = 1;
```

```
reg_curve(j);
↦ // Ideal i of S defining a projective curve C in P3:
↦ //   - i is saturated: YES
↦ //   - C is arithm. Cohen-Macaulay: NO
↦ //   - reg(C) attained at the last step of a m.g.f.r. of i-sat: YES
↦ //   - regularity of the Hilbert function of S/i-sat: 92
↦ //   - time for computing reg(C): 0 sec.
↦ // Castelnuovo-Mumford regularity of C:
↦ 93
```

D.4.5.3 reg_moncurve

Procedure from library `mregular.lib` (see Section D.4.5 [mregular_lib], page 404).

Usage: reg_moncurve (a0,...,an) ; ai integers with a0=0 < a1 < ... < an=:d

Return: an integer, the Castelnuovo-Mumford regularity of the projective monomial curve C in Pn parametrically defined by:
x(0)=t^d , x(1)=s^(a1)t^(d-a1), ... , x(n)=s^d.

Assume: a0=0 < a1 < ... < an are integers and the base field is infinite.

Note: The defining ideal I(C) in S is determined using elimination. The procedure reg_curve is improved in this case since one knows beforehand that the dimension is 2, that the variables are in Noether position, that I(C) is prime.
If printlevel > 0 (default = 0) additional information is displayed. In particular, says if C is arithmetically Cohen-Macaulay or not, determines in which step of a minimal graded free resolution of I(C) the regularity is attained, and sometimes gives the value of the regularity of the Hilbert function of S/I(C) (otherwise, an upper bound is given).

Example:

```
LIB "mregular.lib";
// The 1st example is the twisted cubic:
reg_moncurve(0,1,2,3);
↦ 2
// The 2nd. example is the non arithm. Cohen-Macaulay monomial curve in P4
// parametrized by: x(0)-s6,x(1)-s5t,x(2)-s3t3,x(3)-st5,x(4)-t6:
reg_moncurve(0,1,3,5,6);
↦ 3
// Additional information can be obtained as follows:
printlevel = 1;
reg_moncurve(0,1,3,5,6);
↦ // Sequence of integers defining a monomial curve C in P4:
↦ //   - time for computing ideal I(C) of S (elimination): 0 sec.
↦ //   - C is arithm. Cohen-Macaulay: NO
↦ //   - reg(C) attained at the last step of a m.g.f.r. of I(C): YES
↦ //   - reg(C) attained at the second last step of a m.g.f.r. of I(C): YES
↦ //   - regularity of the Hilbert function of S/I(C): 2
↦ //   - time for computing reg(C): 0 sec.
↦ // Castelnuovo-Mumford regularity of C:
↦ 3
```

D.4.6 normal_lib

Library: normal.lib

Purpose: Normalization of Affine Rings

Authors: G.-M. Greuel, greuel@mathematik.uni-kl.de,
G. Pfister, pfister@mathematik.uni-kl.de

Procedures:

D.4.6.1 normal

Procedure from library `normal.lib` (see Section D.4.6 [normal_lib], page 406).

Assume: The ideal must be radical, for non radical ideals the output may be wrong (i=radical(i); makes i radical)

Return: a list of rings, say nor and in case of choose="wd" an integer
at the end of the list.
each ring nor[i] contains two ideals
with given names norid and normap such that
- the direct sum of the rings nor[i]/norid is
the normalization of basering/id;
- normap gives the normalization map from basering/id
to nor[i]/norid (for each i)

Note: to use the i-th ring type: def R=nor[i]; setring R;.
Increasing printlevel displays more comments (default: printlevel=0)
Not implemented for local or mixed orderings.
If the input ideal i is weighted homogeneous a weighted ordering may be used (qhweight(i); computes weights).

Example:

```
LIB "normal.lib";
ring r=32003,(x,y,z),wp(2,1,2);
ideal i=z3-xy4;
list nor=normal(i);
↦
↦ // 'normal' created a list of 1 ring(s).
↦ // nor[1+1] is the delta-invariant in case of choose=wd.
↦ // To see the rings, type (if the name of your list is nor):
↦      show( nor);
↦ // To access the 1-st ring and map (similar for the others), type:
↦      def R = nor[1]; setring R;  norid; normap;
↦ // R/norid is the 1-st ring of the normalization and
↦ // normap the map from the original basering to R/norid
show(nor);
↦ // list, 1 element(s):
↦ [1]:
↦    // ring: (32003),(T(1),T(2),T(3)),(a(2,1,1),dp(3),C);
↦    // minpoly = 0
↦ // objects belonging to this ring:
↦ // normap               [0]  ideal, 3 generator(s)
↦ // norid                [0]  ideal, 1 generator(s)
def r1=nor[1];
setring r1;
norid;
↦ norid[1]=T(3)3-T(1)T(2)
```

```
normap;
↦ normap[1]=T(1)
↦ normap[2]=T(2)
↦ normap[3]=T(2)T(3)
ring s=0,(x,y),dp;
ideal i=(x-y^2)^2 - y*x^3;
nor=normal(i,"wd");
↦
↦ // 'normal' created a list of 1 ring(s).
↦ // nor[1+1] is the delta-invariant in case of choose=wd.
↦ // To see the rings, type (if the name of your list is nor):
↦      show( nor);
↦ // To access the 1-st ring and map (similar for the others), type:
↦      def R = nor[1]; setring R;  norid; normap;
↦ // R/norid is the 1-st ring of the normalization and
↦ // normap the map from the original basering to R/norid
//the delta-invariant
nor[size(nor)];
↦ 3
```

D.4.6.2 HomJJ

Procedure from library `normal.lib` (see Section D.4.6 [normal_lib], page 406).

Usage: HomJJ (Li); Li = list: ideal SBid, ideal id, ideal J, poly p

Assume: R = P/id, P = basering, a polynomial ring, id an ideal of P,
SBid = standard basis of id,
J = ideal of P containing the polynomial p,
p = nonzero divisor of R

Compute: Endomorphism ring End_R(J)=Hom_R(J,J) with its ring structure as affine ring, together with the canonical map R –> Hom_R(J,J), where R is the quotient ring of P modulo the standard basis SBid.

Return: a list l of two objects

l[1] : a polynomial ring, containing two ideals, 'endid' and 'endphi'
such that l[1]/endid = Hom_R(J,J) and
endphi describes the canonical map R -> Hom_R(J,J)
l[2] : an integer which is 1 if phi is an isomorphism, 0 if not
l[3] : an integer, the contribution to delta

Note: printlevel >=1: display comments (default: printlevel=0)

Example:

```
LIB "normal.lib";
ring r   = 0,(x,y),wp(2,3);
ideal id = y^2-x^3;
ideal J  = x,y;
poly p   = x;
list Li = std(id),id,J,p;
list L   = HomJJ(Li);
def end = L[1];    // defines ring L[1], containing ideals endid, endphi
setring end;       // makes end the basering
end;
```

```
↦ //   characteristic : 0
↦ //   number of vars : 1
↦ //        block   1 : ordering dp
↦ //                  : names    T(1)
↦ //        block   2 : ordering C
endid;             // end/endid is isomorphic to End(r/id) as ring
↦ endid[1]=0
map psi = r,endphi;// defines the canonical map r/id -> End(r/id)
psi;
↦ psi[1]=T(1)^2
↦ psi[2]=T(1)^3
```

D.4.6.3 genus

Procedure from library `normal.lib` (see Section D.4.6 [normal_lib], page 406).

Usage: genus(I) or genus(i,1); I a 1-dimensional ideal

Return: an integer, the geometric genus p_g = p_a - delta of the projective curve defined by I, where p_a is the arithmetic genus.

Note: delta is the sum of all local delta-invariants of the singularities, i.e. dim(R'/R), R' the normalization of the local ring R of the singularity.
genus(i,1) uses the normalization to compute delta. Usually this is slow but sometimes not.

Example:

```
LIB "normal.lib";
ring r=0,(x,y),dp;
ideal i=y^9 - x^2*(x - 1)^9;
genus(i);
↦ 0
```

D.4.7 primdec_lib

Library: primdec.lib

Purpose: Primary Decomposition and Radical of Ideals

Authors: Gerhard Pfister, pfister@mathematik.uni-kl.de (GTZ)
Wolfram Decker, decker@math.uni-sb.de (SY)
Hans Schoenemann, hannes@mathematik.uni-kl.de (SY)

Overview: Algorithms for primary decomposition based on the ideas of Gianni, Trager and Zacharias (implementation by Gerhard Pfister), respectively based on the ideas of Shimoyama and Yokoyama (implementation by Wolfram Decker and Hans Schoenemann).
The procedures are implemented to be used in characteristic 0.
They also work in positive characteristic >> 0.
In small characteristic and for algebraic extensions, primdecGTZ and minAssGTZ may not terminate.
Algorithms for the computation of the radical based on the ideas of Krick, Logar and Kemper (implementation by Gerhard Pfister).

Procedures:

D.4.7.1 Ann

Procedure from library `primdec.lib` (see Section D.4.7 [primdec_lib], page 409).

D.4.7.2 primdecGTZ

Procedure from library `primdec.lib` (see Section D.4.7 [primdec_lib], page 409).

Usage: primdecGTZ(i); i ideal

Return: a list pr of primary ideals and their associated primes:

pr[i][1] the i-th primary component,
pr[i][2] the i-th prime component.

Note: Algorithm of Gianni/Trager/Zacharias.
Designed for characteristic 0, works also in char k > 0, if it terminates (may result in an infinite loop in small characteristic!)

Example:

```
LIB "primdec.lib";
ring  r = 0,(x,y,z),lp;
poly  p = z2+1;
poly  q = z3+2;
ideal i = p*q^2,y-z2;
list pr = primdecGTZ(i);
pr;
↦ [1]:
↦    [1]:
↦       _[1]=z2+1
↦       _[2]=y-z2
↦    [2]:
↦       _[1]=z2+1
↦       _[2]=y-z2
↦ [2]:
↦    [1]:
↦       _[1]=z6+4z3+4
↦       _[2]=y-z2
↦    [2]:
↦       _[1]=z3+2
↦       _[2]=y-z2
```

D.4.7.3 primdecSY

Procedure from library `primdec.lib` (see Section D.4.7 [primdec_lib], page 409).

Usage: primdecSY(i); i ideal, c int

Return: a list pr of primary ideals and their associated primes:

pr[i][1] the i-th primary component,
pr[i][2] the i-th prime component.

Note: Algorithm of Shimoyama/Yokoyama.

if c=0, the given ordering of the variables is used,
if c=1, minAssChar tries to use an optimal ordering,
if c=2, minAssGTZ is used,
if c=3, minAssGTZ and facstd are used.

Example:

```
LIB "primdec.lib";
ring  r = 0,(x,y,z),lp;
poly  p = z2+1;
poly  q = z3+2;
ideal i = p*q^2,y-z2;
list pr = primdecSY(i);
pr;
↦ [1]:
↦    [1]:
↦       _[1]=z2+1
↦       _[2]=y+1
↦    [2]:
↦       _[1]=z2+1
↦       _[2]=y+1
↦ [2]:
↦    [1]:
↦       _[1]=z6+4z3+4
↦       _[2]=y-z2
↦    [2]:
↦       _[1]=z3+2
↦       _[2]=y-z2
```

D.4.7.4 minAssGTZ

Procedure from library `primdec.lib` (see Section D.4.7 [primdec_lib], page 409).

Usage: minAssGTZ(i); i ideal
minAssGTZ(i,1); i ideal does not use the factorizing Groebner

Return: a list, the minimal associated prime ideals of i.

Note: Designed for characteristic 0, works also in char k > 0 if it terminates, may result in an infinite loop in small characteristic.

Example:

```
LIB "primdec.lib";
ring  r = 0,(x,y,z),dp;
poly  p = z2+1;
poly  q = z3+2;
ideal i = p*q^2,y-z2;
list pr = minAssGTZ(i);
pr;
↦ [1]:
↦    _[1]=z3+2
↦    _[2]=-z2+y
↦ [2]:
↦    _[1]=z2+1
↦    _[2]=-z2+y
```

D.4.7.5 minAssChar

Procedure from library `primdec.lib` (see Section D.4.7 [primdec_lib], page 409).

Usage: minAssChar(i[,c]); i ideal, c int.

Return: list, the minimal associated prime ideals of i.

Note: If c=0, the given ordering of the variables is used.
Otherwise, the system tries to find an optimal ordering, which in some cases may considerably speed up the algorithm.
Due to a bug in the factorization, the result may be not completely decomposed in small characteristic.

Example:

```
LIB "primdec.lib";
ring  r = 0,(x,y,z),dp;
poly  p = z2+1;
poly  q = z3+2;
ideal i = p*q^2,y-z2;
list pr = minAssChar(i);
pr;
↦ [1]:
↦    _[1]=z2-y
↦    _[2]=yz+2
↦    _[3]=y2+2z
↦ [2]:
↦    _[1]=y+1
↦    _[2]=z2+1
```

D.4.7.6 testPrimary

Procedure from library `primdec.lib` (see Section D.4.7 [primdec_lib], page 409).

Usage: testPrimary(pr,k); pr a list, k an ideal.

Assume: pr is the result of primdecGTZ(k) or primdecSY(k).

Return: int, 1 if the intersection of the ideals in pr is k, 0 if not

Example:

```
LIB "primdec.lib";
ring  r = 32003,(x,y,z),dp;
poly  p = z2+1;
poly  q = z4+2;
ideal i = p^2*q^3,(y-z3)^3,(x-yz+z4)^4;
list pr = primdecGTZ(i);
testPrimary(pr,i);
↦ 1
```

D.4.7.7 radical

Procedure from library `primdec.lib` (see Section D.4.7 [primdec_lib], page 409).

Usage: radical(i); i ideal.

Return: ideal, the radical of i.

Note: A combination of the algorithms of Krick/Logar and Kemper is used. Works also in positive characteristic (Kempers algorithm).

Example:

```
LIB "primdec.lib";
ring  r = 0,(x,y,z),dp;
poly  p = z2+1;
poly  q = z3+2;
ideal i = p*q^2,y-z2;
ideal pr= radical(i);
pr;
↦ pr[1]=z2-y
↦ pr[2]=y2z+yz+2y+2
```

D.4.7.8 radicalEHV

Procedure from library `primdec.lib` (see Section D.4.7 [primdec_lib], page 409).

Usage: radicalEHV(i); i ideal.

Return: ideal, the radical of i.

Note: Uses the algorithm of Eisenbud/Huneke/Vasconcelos, which reduces the computation to the complete intersection case, by taking, in the general case, a generic linear combination of the input.
Works only in characteristic 0 or p large.

Example:

```
LIB "primdec.lib";
ring  r = 0,(x,y,z),dp;
poly  p = z2+1;
poly  q = z3+2;
ideal i = p*q^2,y-z2;
ideal pr= radicalEHV(i);
pr;
↦ pr[1]=z2-y
↦ pr[2]=y2z+yz+2y+2
↦ pr[3]=y3+y2+2yz+2z
```

D.4.7.9 equiRadical

Procedure from library `primdec.lib` (see Section D.4.7 [primdec_lib], page 409).

Usage: equiRadical(i); i ideal

Return: ideal, intersection of associated primes of i of maximal dimension.

Note: A combination of the algorithms of Krick/Logar and Kemper is used. Works also in positive characteristic (Kempers algorithm).

Example:

```
LIB "primdec.lib";
ring  r = 0,(x,y,z),dp;
poly  p = z2+1;
poly  q = z3+2;
ideal i = p*q^2,y-z2;
ideal pr= equiRadical(i);
pr;
↦ pr[1]=z2-y
↦ pr[2]=y2z+yz+2y+2
```

D.4.7.10 prepareAss

Procedure from library `primdec.lib` (see Section D.4.7 [primdec_lib], page 409).

Usage: prepareAss(i); i ideal

Return: list, the radicals of the maximal dimensional components of i.

Note: Uses algorithm of Eisenbud/Huneke/Vasconcelos.

Example:

```
LIB "primdec.lib";
ring  r = 0,(x,y,z),dp;
poly  p = z2+1;
poly  q = z3+2;
ideal i = p*q^2,y-z2;
list pr = prepareAss(i);
pr;
↦ [1]:
↦    _[1]=z2-y
↦    _[2]=y2z+yz+2y+2
```

D.4.7.11 equidim

Procedure from library `primdec.lib` (see Section D.4.7 [primdec_lib], page 409).

Usage: equidim(i) or equidim(i,1) ; i ideal

Return: list of equidimensional ideals a[1],...,a[s] with:
- a[s] the equidimensional locus of i, i.e. the intersection of the primary ideals of dimension of i
- a[1],...,a[s-1] the lower dimensional equidimensional loci.

Note: An embedded component q (primary ideal) of i can be replaced in the decomposition by a primary ideal q1 with the same radical as q.
`equidim(i,1)` uses the algorithm of Eisenbud/Huneke/Vasconcelos.

Example:

```
LIB "primdec.lib";
ring  r = 32003,(x,y,z),dp;
ideal i = intersect(ideal(z),ideal(x,y),ideal(x2,z2),ideal(x5,y5,z5));
equidim(i);
↦ [1]:
↦    _[1]=z4
↦    _[2]=y5
↦    _[3]=x5
↦    _[4]=x3z3
↦    _[5]=x4y4
↦ [2]:
↦    _[1]=yz
↦    _[2]=xz
↦    _[3]=x2
↦ [3]:
↦    _[1]=z
```

D.4.7.12 equidimMax

Procedure from library `primdec.lib` (see Section D.4.7 [primdec_lib], page 409).

Usage: equidimMax(i); i ideal

Return: ideal of equidimensional locus (of maximal dimension) of i.

Example:

```
LIB "primdec.lib";
ring  r = 32003,(x,y,z),dp;
ideal i = intersect(ideal(z),ideal(x,y),ideal(x2,z2),ideal(x5,y5,z5));
equidimMax(i);
↦ _[1]=z
```

D.4.7.13 equidimMaxEHV

Procedure from library `primdec.lib` (see Section D.4.7 [primdec_lib], page 409).

Usage: equidimMaxEHV(i); i ideal

Return: ideal, the equidimensional component (of maximal dimension) of i.

Note: Uses algorithm of Eisenbud, Huneke and Vasconcelos.

Example:

```
LIB "primdec.lib";
ring  r = 0,(x,y,z),dp;
ideal i=intersect(ideal(z),ideal(x,y),ideal(x2,z2),ideal(x5,y5,z5));
equidimMaxEHV(i);
↦ _[1]=z
```

D.4.7.14 zerodec

Procedure from library `primdec.lib` (see Section D.4.7 [primdec_lib], page 409).

Usage: zerodec(I); I ideal

Assume: I is zero-dimensional, the characteristic of the ground field is 0

Return: list of primary ideals, the zero-dimensional decomposition of I

Note: The algorithm (of Monico), works well only for a small total number of solutions (`vdim(std(I))` should be < 100) and without parameters. In practice, it works also in large characteristic p>0 but may fail for small p.
If printlevel > 0 (default = 0) additional information is displayed.

Example:

```
LIB "primdec.lib";
ring r  = 0,(x,y),dp;
ideal i = x2-2,y2-2;
list pr = zerodec(i);
pr;
↦ [1]:
↦    _[1]=y2-2
↦    _[2]=xy+2
↦    _[3]=x2-2
↦ [2]:
↦    _[1]=y2-2
↦    _[2]=xy-2
↦    _[3]=x2-2
```

D.4.8 primitiv_lib

Library: primitiv.lib

Purpose: Computing a Primitive Element

Author: Martin Lamm, email: lamm@mathematik.uni-kl.de

Procedures:

D.4.8.1 primitive

Procedure from library `primitiv.lib` (see Section D.4.8 [primitiv_lib], page 416).

Usage: primitive(i); i ideal

Assume: i is given by generators m[1],...,m[n] such that for j=1,...,n
- m[j] is a polynomial in k[x(1),...,x(j)]
- m[j](a[1],...,a[j-1],x(j)) is the minimal polynomial for a[j] over k(a[1],...,a[j-1])
(k the ground field of the current basering and x(1),...,x(n) the ring variables).

Return: ideal j in k[x(n)] with
- j[1] a minimal polynomial for a primitive element b of k(a[1],...,a[n]) over k,
- j[2],...,j[n+1] polynomials in k[x(n)] such that j[i+1](b)=a[i] for i=1,...,n.

Note: the number of variables in the basering has to be exactly n, the number of given generators (i.e., minimal polynomials).
If the ground field k has only a few elements it may happen that no linear combination of a[1],...,a[n] is a primitive element. In this case `primitive(i)` returns the zero ideal, and one should use `primitive_extra(i)` instead.

Example:

```
LIB "primitiv.lib";
ring exring=0,(x,y),dp;
ideal i=x2+1,y2-x;                  // compute Q(i,i^(1/2))=:L
ideal j=primitive(i);
j[1];                               // L=Q(a) with a=(-1)^(1/4)
↦ y4+1
j[2];                               // i=a^2
↦ y2
j[3];                               // i^(1/2)=a
↦ y
// the 2nd element was already primitive!
j=primitive(ideal(x2-2,y2-3));      // compute Q(sqrt(2),sqrt(3))
j[1];
↦ y4-10y2+1
j[2];
↦ 1/2y3-9/2y
j[3];
↦ -1/2y3+11/2y
// no element was primitive -- the calculation of primitive elements
// is based on a random choice.
```

See also: Section D.4.8.2 [primitive_extra], page 417.

D.4.8.2 primitive_extra

Procedure from library `primitiv.lib` (see Section D.4.8 [primitiv_lib], page 416).

Usage: primitive_extra(i); i ideal

Assume: The ground field of the basering is k=Q or k=Z/pZ and the ideal i is given by 2 generators f,g with the following properties:

f is the minimal polynomial of a in k[x],
g is a polynomial in k[x,y] s.th. g(a,y) is the minpoly of b in k(a)[y].

Here, x is the name of the first ring variable, y the name of the second.

Return: ideal j in k[y] such that

j[1] is the minimal polynomial for a primitive element c of k(a,b) over k,
j[2] is a polynomial s.th. j[2](c)=a.

Note: While `primitive(i)` may fail for finite fields, `primitive_extra(i)` tries all elements of k(a,b) and, hence, always finds a primitive element.
In order to do this (try all elements), field extensions like Z/pZ(a) are not allowed for the ground field k.
`primitive_extra(i)` assumes that the second generator, g, is monic as polynomial in (k[x])[y].

Example:

```
LIB "primitiv.lib";
ring exring=3,(x,y),dp;
ideal i=x2+1,y3+y2-1;
primitive_extra(i);
↦ _[1]=y6-y5+y4-y3-y-1
↦ _[2]=y5+y4+y2+y+1
ring extension=(3,y),x,dp;
minpoly=y6-y5+y4-y3-y-1;
number a=y5+y4+y2+y+1;
a^2;
↦ -1
factorize(x2+1);
↦ [1]:
↦    _[1]=1
↦    _[2]=x+(y5+y4+y2+y+1)
↦    _[3]=x+(-y5-y4-y2-y-1)
↦ [2]:
↦    1,1,1
factorize(x3+x2-1);
↦ [1]:
↦    _[1]=1
↦    _[2]=x+(y3+y+1)
↦    _[3]=x+(y5+y4+y2+1)
↦    _[4]=x+(-y5-y4-y3-y2-y-1)
↦ [2]:
↦    1,1,1,1
```

D.4.8.3 splitring

Procedure from library `primitiv.lib` (see Section D.4.8 [primitiv_lib], page 416).

Usage: splitring(f,R[,L]); f poly, R string, L list of polys and/or ideals (optional)

Assume: f is univariate and irreducible over the active basering.
The active ring must allow an algebraic extension (e.g., it cannot be a transcendent ring extension of Q or Z/p).

Create: a ring with name R, in which f is reducible, and CHANGE to it.

Return: list L mapped into the new ring R, if L is given; else nothing

Note: If the old ring has no parameter, the name `a` is chosen for the parameter of R (if `a` is no ring variable; if it is, `b` is chosen, etc.; if `a,b,c,o` are ring variables, `splitring(f,R[,L])` produces an error message), otherwise the name of the parameter is kept and only the minimal polynomial is changed.
The names of the ring variables and the orderings are not affected.
It is also allowed to call `splitring` with R="". Then the old basering will be REPLACED by the new ring (with the same name as the old ring).

Example:

```
LIB "primitiv.lib";
ring r=0,(x,y),dp;
splitring(x2-2,"r1");   // change to Q(sqrt(2))
// change to Q(sqrt(2),sqrt(sqrt(2)))=Q(a) and return the transformed
// old parameter:
splitring(x2-a,"r2",a);
↦ // new minimal polynomial: a4-2
↦ [1]:
↦    (a2)
// the result is (a)^2 = (sqrt(sqrt(2)))^2
nameof(basering);
↦ r2
r2;
↦ //   characteristic : 0
↦ //   1 parameter    : a
↦ //   minpoly        : (a4-2)
↦ //   number of vars : 2
↦ //        block   1 : ordering dp
↦ //                  : names    x y
↦ //        block   2 : ordering C
kill r1; kill r2;
```

D.4.9 reesclos_lib

Library: reesclos.lib

Purpose: procedures to compute the int. closure of an ideal

Author: Tobias Hirsch, email: hirsch@math.tu-cottbus.de

Overview: A library to compute the integral closure of an ideal I in a polynomial ring R=K[x(1),...,x(n)] using the Rees Algebra R[It] of I. It computes the integral closure of R[It] (in the same manner as done in the library 'normal.lib'), which is a graded subalgebra of R[t]. The degree-k-component is the integral closure of the k-th power of I.
These procedures can also be used to compute the integral closure R^ of an integral domain R=k[x(1),...,x(n)]/ker, ker a prime ideal, in its quotient field K=Q(R), as

an affine ring R^=k[T(1),...,T(s)]]/J and to get representations of elements of R^ as fractions of elements of R.

Procedures:

D.4.9.1 ReesAlgebra

Procedure from library `reesclos.lib` (see Section D.4.9 [reesclos_lib], page 418).

Usage: ReesAlgebra (I); I = ideal

Return: The Rees algebra R[It] as an affine ring, where I is an ideal in R. The procedure returns a list containing two rings:
[1]: a ring, say RR; in the ring an ideal ker such that R[It]=RR/ker

[2]: a ring, say Kxt; the basering with additional variable t containing an ideal mapI that defines the map RR–>Kxt

Example:

```
LIB "reesclos.lib";
ring R = 0,(x,y),dp;
ideal I = x2,xy4,y5;
list L = ReesAlgebra(I);
def Rees = L[1];       // defines the ring Rees, containing the ideal ker
setring Rees;          // passes to the ring Rees
Rees;
↦ //   characteristic : 0
↦ //   number of vars : 5
↦ //        block   1 : ordering dp
↦ //                  : names    x y U(1) U(2) U(3)
↦ //        block   2 : ordering C
ker;                   // R[It] is isomorphic to Rees/ker
↦ ker[1]=y*U(2)-x*U(3)
↦ ker[2]=y^3*U(1)*U(3)-U(2)^2
↦ ker[3]=y^4*U(1)-x*U(2)
↦ ker[4]=x*y^2*U(1)*U(3)^2-U(2)^3
↦ ker[5]=x^2*y*U(1)*U(3)^3-U(2)^4
↦ ker[6]=x^3*U(1)*U(3)^4-U(2)^5
```

D.4.9.2 normalI

Procedure from library `reesclos.lib` (see Section D.4.9 [reesclos_lib], page 418).

Usage: normalI(I [,p[,c]]); I an ideal, p and c optional integers

Return: the integral closure of I,...,I^p. If p is not given, or p==0, compute the closure of all powers up to the maximum degree in t occurring in the generators of the closure of R[It] (so this is the last one that is not just the sum/product of the above ones). c is transferred to the procedure primeClosure and toggles its behavior in computing the integral closure of R[It].
The result is a list containing the closure of the desired powers of I as ideals of the basering.

Example:

```
LIB "reesclos.lib";
ring R=0,(x,y),dp;
```

```
ideal I = x2,xy4,y5;
list J = normalI(I);
I;
↦ I[1]=x2
↦ I[2]=xy4
↦ I[3]=y5
J;                              // J[1] is the integral closure of I
↦ [1]:
↦    _[1]=x2
↦    _[2]=y5
↦    _[3]=-xy3
```

D.4.9.3 primeClosure

Procedure from library `reesclos.lib` (see Section D.4.9 [reesclos_lib], page 418).

Usage: primeClosure(L [,c]); L a list of a ring containing a prime ideal ker, c an optional integer

Return: a list L consisting of rings L[1],...,L[n] such that
- L[1] is a copy of (not a reference to!) the input ring L[1] - all rings L[i] contain ideals ker, L[2],...,L[n] contain ideals phi such that
L[1]/ker –> ... –> L[n]/ker
are injections given by the corresponding ideals phi, and L[n]/ker is the integral closure of L[1]/ker in its quotient field. - all rings L[i] contain a polynomial nzd such that elements of L[i]/ker are quotients of elements of L[i-1]/ker with denominator nzd via the injection phi.

Note: - L is constructed by recursive calls of primeClosure itself. - c determines the choice of nzd:
- c not given or equal to 0: first generator of the ideal SL, the singular locus of Spec(L[i]/ker)
- c<>0: the generator of SL with least number of monomials.

Example:

```
LIB "reesclos.lib";
ring R=0,(x,y),dp;
ideal I=x4,y4;
def K=ReesAlgebra(I)[1];        // K contains ker such that K/ker=R[It]
list L=primeClosure(K);
def R(1)=L[1];                  // L[4] contains ker, L[4]/ker is the
def R(4)=L[4];                  // integral closure of L[1]/ker
setring R(1);
R(1);
↦ //   characteristic : 0
↦ //   number of vars : 4
↦ //        block   1 : ordering dp
↦ //                  : names    x y U(1) U(2)
↦ //        block   2 : ordering C
ker;
↦ ker[1]=y^4*U(1)-x^4*U(2)
setring R(4);
R(4);
↦ //   characteristic : 0
↦ //   number of vars : 7
```

```
↦ //        block   1 : ordering a
↦ //                  : names    T(1) T(2) T(3) T(4) T(5) T(6) T(7)
↦ //                  : weights     1    1    1    1    1    1    1
↦ //        block   2 : ordering dp
↦ //                  : names    T(1) T(2) T(3) T(4) T(5) T(6) T(7)
↦ //        block   3 : ordering C
ker;
↦ ker[1]=T(2)*T(5)-T(1)*T(7)
↦ ker[2]=T(5)*T(6)-T(3)*T(7)
↦ ker[3]=T(4)*T(6)-T(5)*T(7)
↦ ker[4]=T(5)^2-T(6)*T(7)
↦ ker[5]=T(4)*T(5)-T(7)^2
↦ ker[6]=T(3)*T(5)-T(6)^2
↦ ker[7]=T(1)*T(5)-T(2)*T(6)
↦ ker[8]=T(3)*T(4)-T(6)*T(7)
↦ ker[9]=T(1)*T(4)-T(2)*T(7)
↦ ker[10]=T(2)*T(3)-T(1)*T(6)
↦ ker[11]=T(2)^2*T(6)-T(1)^2*T(7)
```

D.4.9.4 closureRingtower

Procedure from library `reesclos.lib` (see Section D.4.9 [reesclos_lib], page 418).

Usage: closureRingtower(list L); L a list of rings

Create: rings R(1),...,R(n) such that R(i)=L[i] for all i

Example:

```
LIB "reesclos.lib";
ring R=0,(x,y),dp;
ideal I=x4,y4;
list L=primeClosure(ReesAlgebra(I)[1]);
closureRingtower(L);
R(1);
↦ //   characteristic : 0
↦ //   number of vars : 4
↦ //        block   1 : ordering dp
↦ //                  : names    x y U(1) U(2)
↦ //        block   2 : ordering C
R(4);
↦ //   characteristic : 0
↦ //   number of vars : 7
↦ //        block   1 : ordering a
↦ //                  : names    T(1) T(2) T(3) T(4) T(5) T(6) T(7)
↦ //                  : weights     1    1    1    1    1    1    1
↦ //        block   2 : ordering dp
↦ //                  : names    T(1) T(2) T(3) T(4) T(5) T(6) T(7)
↦ //        block   3 : ordering C
```

D.4.9.5 closureFrac

Procedure from library `reesclos.lib` (see Section D.4.9 [reesclos_lib], page 418).

Create: a list fraction of two elements of L[1], such that
f=fraction[1]/fraction[2] via the injections phi L[i]–>L[i+1].

Example:

```
LIB "reesclos.lib";
ring R=0,(x,y),dp;
ideal ker=x2+y2;
export R;
↦ // ** ‘R‘ is already global
list L=primeClosure(R);          // We normalize R/ker
closureRingtower(L);             // Now R/ker=R(1) with normalization R(2)
setring R(2);
kill(R);
phi;                             // The map R(1)-->R(2)
↦ phi[1]=T(1)*T(2)
↦ phi[2]=T(1)
poly f=T(1)*T(2);                // We will get a representation of f
export R(2);
↦ // ** ‘R(2)‘ is already global
closureFrac(L);
setring R(1);
kill (R(2));
fraction;                        // f=fraction[1]/fraction[2] via phi
↦ [1]:
↦    xy
↦ [2]:
↦    y
```

D.4.10 intprog_lib

Library: intprog.lib

Purpose: Integer Programming with Groebner Basis Methods

Author: Christine Theis, email: ctheis@math.uni-sb.de

Procedures:

D.4.10.1 solve_IP

Procedure from library `intprog.lib` (see Section D.4.10 [intprog_lib], page 422).

Usage: solve_IP(A,bx,c,alg); A intmat, bx intvec, c intvec, alg string. solve_IP(A,bx,c,alg); A intmat, bx list of intvec, c intvec, alg string.
solve_IP(A,bx,c,alg,prsv); A intmat, bx intvec, c intvec, alg string, prsv intvec.
solve_IP(A,bx,c,alg,prsv); A intmat, bx list of intvec, c intvec, alg string, prsv intvec.

Return: same type as bx: solution of the associated integer programming problem(s) as explained in

Section C.6 [Toric ideals and integer programming], page 287.

Note: This procedure returns the solution(s) of the given IP-problem(s) or the message ‘not solvable’.
One may call the procedure with several different algorithms:
- the algorithm of Conti/Traverso (ct),
- the positive variant of the algorithm of Conti/Traverso (pct),
- the algorithm of Conti/Traverso using elimination (ect),
- the algorithm of Pottier (pt),

- an algorithm of Bigatti/La Scala/Robbiano (blr),
- the algorithm of Hosten/Sturmfels (hs),
- the algorithm of DiBiase/Urbanke (du). The argument ‘alg’ should be the abbreviation for an algorithm as above: ct, pct, ect, pt, blr, hs or du.

‘ct’ allows computation of an optimal solution of the IP-problem directly from the right-hand vector b.
The same is true for its ‘positive’ variant ‘pct’ which may only be applied if A and b have nonnegative entries.
All other algorithms need initial solutions of the IP-problem.

If ‘alg’ is chosen to be ‘ct’ or ‘pct’, bx is read as the right hand vector b of the system Ax=b. b should then be an intvec of size m where m is the number of rows of A.
Furthermore, bx and A should be nonnegative if ‘pct’ is used. If ‘alg’ is chosen to be ‘ect’,‘pt’,‘blr’,‘hs’ or ‘du’, bx is read as an initial solution x of the system Ax=b. bx should then be a nonnegative intvec of size n where n is the number of columns of A.

If ‘alg’ is chosen to be ‘blr’ or ‘hs’, the algorithm needs a vector with positive coefficients in the row space of A.
If no row of A contains only positive entries, one has to use the versions of solve_IP which take such a vector prsv as an argument.

solve_IP may also be called with a list bx of intvecs instead of a single intvec.

Example:

```
LIB "intprog.lib";
// 1. call with single right-hand vector
intmat A[2][3]=1,1,0,0,1,1;
intvec b1=1,1;
intvec c=2,2,1;
intvec solution_vector=solve_IP(A,b1,c,"pct");
solution_vector;"";
↦ 0,1,0
↦
// 2. call with list of right-hand vectors
intvec b2=-1,1;
list l=b1,b2;
l;
↦ [1]:
↦    1,1
↦ [2]:
↦    -1,1
list solution_list=solve_IP(A,l,c,"ct");
solution_list;"";
↦ [1]:
↦    0,1,0
↦ [2]:
↦    not solvable
↦
// 3. call with single initial solution vector
A=2,1,-1,-1,1,2;
b1=3,4,5;
solve_IP(A,b1,c,"du");"";
↦ 0,7,2
↦
// 4. call with single initial solution vector
```

```
//    and algorithm needing a positive row space vector
solution_vector=solve_IP(A,b1,c,"hs");"";
↦ ERROR: The chosen algorithm needs a positive vector in the row space of t\
   he matrix.
↦ 0
↦
// 5. call with single initial solution vector
//     and positive row space vector
intvec prsv=1,2,1;
solution_vector=solve_IP(A,b1,c,"hs",prsv);
solution_vector;"";
↦ 0,7,2
↦
// 6. call with list of initial solution vectors
//    and positive row space vector
b2=7,8,0;
l=b1,b2;
l;
↦ [1]:
↦    3,4,5
↦ [2]:
↦    7,8,0
solution_list=solve_IP(A,l,c,"blr",prsv);
solution_list;
↦ [1]:
↦    0,7,2
↦ [2]:
↦    7,8,0
```

See also: Section C.6.4 [Integer programming], page 290; Section D.4.10 [intprog_lib], page 422; Section D.4.11 [toric_lib], page 424.

D.4.11 toric_lib

Library: toric.lib

Purpose: Standard Basis of Toric Ideals

Author: Christine Theis, email: ctheis@math.uni-sb.de

Procedures:

D.4.11.1 toric_ideal

Procedure from library `toric.lib` (see Section D.4.11 [toric_lib], page 424).

Usage: toric_ideal(A,alg); A intmat, alg string
toric_ideal(A,alg,prsv); A intmat, alg string, prsv intvec

Return: ideal: standard basis of the toric ideal of A

Note: These procedures return the standard basis of the toric ideal of A with respect to the term ordering in the current basering. Not all term orderings are supported: The usual global term orderings may be used, but no block orderings combining them.
One may call the procedure with several different algorithms:
- the algorithm of Conti/Traverso using elimination (ect),

- the algorithm of Pottier (pt),
- an algorithm of Bigatti/La Scala/Robbiano (blr),
- the algorithm of Hosten/Sturmfels (hs),
- the algorithm of DiBiase/Urbanke (du).

The argument 'alg' should be the abbreviation for an algorithm as above: ect, pt, blr, hs or du.

If 'alg' is chosen to be 'blr' or 'hs', the algorithm needs a vector with positive coefficients in the row space of A.
If no row of A contains only positive entries, one has to use the second version of toric_ideal which takes such a vector as its third argument.
For the mathematical background, see

Section C.6 [Toric ideals and integer programming], page 287.

Example:

```
LIB "toric.lib";
ring r=0,(x,y,z),dp;
// call with two arguments
intmat A[2][3]=1,1,0,0,1,1;
A;
↦ 1,1,0,
↦ 0,1,1
ideal I=toric_ideal(A,"du");
I;
↦ I[1]=xz-y
I=toric_ideal(A,"blr");
↦ ERROR: The chosen algorithm needs a positive vector in the row space of t\
   he matrix.
I;
↦ I[1]=0
// call with three arguments
intvec prsv=1,2,1;
I=toric_ideal(A,"blr",prsv);
I;
↦ I[1]=xz-y
```

See also: Section C.6.1 [Toric ideals], page 287; Section D.4.10 [intprog_lib], page 422; Section D.4.11 [toric_lib], page 424; Section D.4.11.2 [toric_std], page 425.

D.4.11.2 toric_std

Procedure from library `toric.lib` (see Section D.4.11 [toric_lib], page 424).

Usage: toric_std(I); I ideal

Return: ideal: standard basis of I

Note: This procedure computes the standard basis of I using a specialized Buchberger algorithm. The generating system by which I is given has to consist of binomials of the form x^u-x^v. There is no real check if I is toric. If I is generated by binomials of the above form, but not toric, toric_std computes an ideal 'between' I and its saturation with respect to all variables.
For the mathematical background, see

Section C.6 [Toric ideals and integer programming], page 287.

Example:

```
LIB "toric.lib";
ring r=0,(x,y,z),wp(3,2,1);
// call with toric ideal (of the matrix A=(1,1,1))
ideal I=x-y,x-z;
ideal J=toric_std(I);
J;
↦ J[1]=y-z
↦ J[2]=x-z
// call with the same ideal, but badly chosen generators:
// 1) not only binomials
I=x-y,2x-y-z;
J=toric_std(I);
↦ ERROR: Generator 2 of the input ideal is no difference of monomials.
// 2) binomials whose monomials are not relatively prime
I=x-y,xy-yz,y-z;
J=toric_std(I);
↦ Warning: The monomials of generator 2 of the input ideal are not relative\
   ly prime.
J;
↦ J[1]=y-z
↦ J[2]=x-z
// call with a non-toric ideal that seems to be toric
I=x-yz,xy-z;
J=toric_std(I);
J;
↦ J[1]=y2-1
↦ J[2]=x-yz
// comparison with real standard basis and saturation
ideal H=std(I);
H;
↦ H[1]=x-yz
↦ H[2]=y2z-z
LIB "elim.lib";
sat(H,xyz);
↦ [1]:
↦    _[1]=x-yz
↦    _[2]=y2-1
↦ [2]:
↦    1
```

See also: Section C.6.1 [Toric ideals], page 287; Section D.4.10 [intprog_lib], page 422; Section D.4.11.1 [toric_ideal], page 424; Section D.4.11 [toric_lib], page 424.

D.5 Singularities

D.5.1 classify_lib

Library: classify.lib

Purpose: Arnold Classifier of Singularities

Authors: Kai Krueger, krueger@mathematik.uni-kl.de
Corina Baciu, baciu@mathematik.uni-kl.de

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

Procedures:

D.5.1.1 basicinvariants

Procedure from library `classify.lib` (see Section D.5.1 [classify_lib], page 426).

Usage: basicinvariants(f); f = poly

Compute: Compute basic invariants of f: an upper bound d for the determinacy, the milnor number mu and the corank c of f

Return: intvec: d, mu, c

Example:

```
LIB "classify.lib";
ring r=0,(x,y,z),ds;
basicinvariants((x2+3y-2z)^2+xyz-(x-y3+x2*z3)^3);
↦ 5,4,2
```

D.5.1.2 classify

Procedure from library `classify.lib` (see Section D.5.1 [classify_lib], page 426).

Usage: classify(f); f=poly

Compute: normal form and singularity type of f with respect to right equivalence, as given in the book "Singularities of differentiable maps, Volume I" by V.I. Arnold, S.M. Gusein-Zade, A.N. Varchenko

Return: normal form of f, of type poly

Remark: This version of classify is only beta. Please send bugs and comments to: "Kai Krueger" <krueger@mathematik.uni-kl.de>
Be sure to have at least Singular version 1.0.1. Updates can be found at:
URL=http://www.mathematik.uni-kl.de/~krueger/Singular/

Note: type init_debug(n); (0 <= n <= 10) in order to get intermediate information, higher values of n give more information.
The proc creates several global objects with names all starting with `@`, hence there should be no name conflicts

Example:

```
LIB "classify.lib";
ring r=0,(x,y,z),ds;
poly f=(x2+3y-2z)^2+xyz-(x-y3+x2*z3)^3;
classify(f);
↦ About the singularity :
↦           Milnor number(f)   = 4
↦           Corank(f)          = 2
↦           Determinacy       <= 5
↦ Guessing type via Milnorcode:   D[k]=D[4]
↦
↦ Computing normal form ...
↦ I have to apply the splitting lemma. This will take some time....:-)
```

```
↦    Arnold step number 4
↦ The singularity
↦    -x3+3/2xy2+1/2x3y-1/16x2y2+3x2y3
↦ is R-equivalent to D[4].
↦    Milnor number = 4
↦    modality      = 0
↦ 2z2+x2y+y3
init_debug(3);
↦ Debugging level change from  0  to  3
classify(f);
↦ Computing Basicinvariants of f ...
↦ About the singularity :
↦           Milnor number(f)   = 4
↦           Corank(f)          = 2
↦           Determinacy       <= 5
↦ Hcode: 1,2,1,0,0
↦ Milnor code :  1,1,1
↦ Debug:(2):  entering HKclass3_teil_1 1,1,1
↦ Debug:(2):  finishing HKclass3_teil_1
↦ Guessing type via Milnorcode:   D[k]=D[4]
↦
↦ Computing normal form ...
↦ I have to apply the splitting lemma. This will take some time....:-)
↦ Debug:(3):  Split the polynomial below using determinacy:  5
↦ Debug:(3):  9y2-12yz+4z2-x3+6x2y-4x2z+xyz+x4+3x2y3
↦ Debug:(2):  Permutations: 3,2,1
↦ Debug:(2):  Permutations: 3,2,1
↦ Debug:(2):  rank determined with Morse rg= 1
↦ Residual singularity f= -x3+3/2xy2+1/2x3y-1/16x2y2+3x2y3
↦ Step 3
↦    Arnold step number 4
↦ The singularity
↦    -x3+3/2xy2+1/2x3y-1/16x2y2+3x2y3
↦ is R-equivalent to D[4].
↦    Milnor number = 4
↦    modality      = 0
↦ Debug:(2):  Decode:
↦ Debug:(2):  S_in= D[4]   s_in= D[4]
↦ Debug:(2):  Looking for Normalform of  D[k] with (k,r,s) = ( 4 , 0 , 0 )
↦ Debug:(2):  Opening Singalarity-database:
↦  DBM: NFlist
↦ Debug:(2):  DBMread( D[k] )= x2y+y^(k-1) .
↦ Debug:(2):  S= f = x2y+y^(k-1);  Tp= x2y+y^(k-1) Key= I_D[k]
↦ Polynom f= x2y+y3   crk= 2   Mu= 4  MlnCd= 1,1,1
↦ Debug:(2):  Info= x2y+y3
↦ Debug:(2):  Normal form NF(f)= 2*x(3)^2+x(1)^2*x(2)+x(2)^3
↦ 2z2+x2y+y3
```

D.5.1.3 corank

Procedure from library `classify.lib` (see Section D.5.1 [classify_lib], page 426).

Usage: corank(f); f=poly

Return: the corank of the Hessian matrix of f, of type int

Remark: corank(f) is the number of variables occurring in the residual singularity after applying 'morsesplit' to f

Example:

```
LIB "classify.lib";
ring r=0,(x,y,z),ds;
poly f=(x2+3y-2z)^2+xyz-(x-y3+x2*z3)^3;
corank(f);
↦ 2
```

D.5.1.4 Hcode

Procedure from library `classify.lib` (see Section D.5.1 [classify_lib], page 426).

Usage: Hcode(v); v=intvec

Return: intvec, coding v according to the number of successive repetitions of an entry

Example:

```
LIB "classify.lib";
intvec v1 = 1,3,5,5,2;
Hcode(v1);
↦ 1,0,1,0,2,0,0,1,0
intvec v2 = 1,2,3,4,4,4,4,4,4,4,3,2,1;
Hcode(v2);
↦ 1,1,1,7,1,1,1
```

D.5.1.5 init_debug

Procedure from library `classify.lib` (see Section D.5.1 [classify_lib], page 426).

Usage: init_debug([level]); level=int

Compute: Set the global variable `@DeBug` to level. The variable `@DeBug` is used by the function debug_log(level, list of strings) to know when to print the list of strings. init_debug() reports only changes of `@DeBug`.

Note: The procedure init_debug(n); is useful as trace-mode. n may range from 0 to 10, higher values of n give more information.

Example:

```
LIB "classify.lib";
init_debug();
debug_log(1,"no trace information printed");
init_debug(1);
↦ Debugging level change from  0  to  1
debug_log(1,"some trace information");
↦ some trace information
init_debug(2);
↦ Debugging level change from  1  to  2
debug_log(2,"nice for debugging scripts");
↦ Debug:(2):  nice for debugging scripts
init_debug(0);
↦ Debugging switched off.
```

D.5.1.6 internalfunctions

Procedure from library `classify.lib` (see Section D.5.1 [classify_lib], page 426).

Usage: internalfunctions();

Return: nothing, display names of internal procedures of classify.lib

Example:

```
LIB "classify.lib";
internalfunctions();
↦    Internal functions for the classification using Arnold's method,
↦    the function numbers correspond to numbers in Arnold's classifier:
↦ Klassifiziere(poly f);      //determine the type of the singularity f
↦   Funktion1bis (poly f, list cstn)
↦   Funktion3 (poly f, list cstn)
↦   Funktion6 (poly f, list cstn)
↦   Funktion13 (poly f, list cstn)
↦   Funktion17 (poly f, list cstn)
↦   Funktion25 (poly f, list cstn)
↦   Funktion40 (poly f, list cstn, int k)
↦   Funktion47 (poly f, list cstn)
↦   Funktion50 (poly f, list cstn)
↦   Funktion58 (poly fin, list cstn)
↦   Funktion59 (poly f, list cstn)
↦   Funktion66 (poly f, list cstn)
↦   Funktion82 (poly f, list cstn)
↦   Funktion83 (poly f, list cstn)
↦   Funktion91 (poly f, list cstn, int k)
↦   Funktion92 (poly f, list cstn, int k)
↦   Funktion93 (poly f, list cstn, int k)
↦   Funktion94 (poly f, list cstn, int k)
↦   Funktion95 (poly f, list cstn, int k)
↦   Funktion96 (poly f, list cstn, int k)
↦   Funktion97 (poly f, list cstn)
↦   Isomorphie_s82_x (poly f, poly fk, int k)
↦   Isomorphie_s82_z (poly f, poly fk, int k)
↦   Isomorphie_s17 (poly f, poly fk, int k, int ct)
↦   printresult (string f,string typ,int Mu,int m,int corank,int K)
↦
↦    Internal functions for the classifcation by invariants:
↦   Cubic (poly f)
↦   parity (int e)             //return the parity of e
↦   HKclass (intvec i)
↦   HKclass3( intvec i, string SG_Typ, int cnt)
↦   HKclass3_teil_1 (intvec i, string SG_Typ, int cnt)
↦   HKclass5 (intvec i, string SG_Typ, int cnt)
↦   HKclass5_teil_1 (intvec i, string SG_Typ, int cnt)
↦   HKclass5_teil_2 (intvec i, string SG_Typ, int cnt)
↦   HKclass7 (intvec i, string SG_Typ, int cnt)
↦   HKclass7_teil_1 (intvec i, string SG_Typ, int cnt)
↦
↦    Internal functions for the Morse-splitting lemma:
↦   Morse(poly fi, int K, int corank)  //splitting lemma itself
↦   Coeffs (list #)
```

```
↦   Coeff
↦
↦    Internal functions providing tools:
↦   ReOrder(poly f)
↦   Singularitaet(string typ,int k,int r,int s,poly a,poly b,poly c,poly d)
↦   RandomPolyK
↦   Faktorisiere(poly f, poly g, int p, int k)   compute g = (ax+by^k)^p
↦   Teile(poly f, poly g);             //divides f by g
↦   GetRf(poly f, int n);
↦   Show(poly f);
↦   checkring();
↦   DecodeNormalFormString(string s);
↦   Setring(int n, string ringname);
↦
```

D.5.1.7 milnorcode

Procedure from library `classify.lib` (see Section D.5.1 [classify_lib], page 426).

Usage: milnorcode(f[,e]); f=poly, e=int

Return: intvec, coding the Hilbert function of the e-th Milnor algebra of f, i.e. of basering/(jacob(f)^e) (default e=1), according to proc Hcode

Example:

```
LIB "classify.lib";
ring r=0,(x,y,z),ds;
poly f=x2y+y3+z2;
milnorcode(f);
↦ 1,1,1
milnorcode(f,2);  // a big second argument may result in memory overflow
↦ 1,0,1,0,2,0,0,1,0
```

D.5.1.8 morsesplit

Procedure from library `classify.lib` (see Section D.5.1 [classify_lib], page 426).

Usage: morsesplit(f); f=poly

Return: Normal form of f in M^3 after application of the splitting lemma

Compute: apply the splitting lemma (generalized Morse lemma) to f

Example:

```
LIB "classify.lib";
ring r=0,(x,y,z),ds;
export r;
↦ // ** `r` is already global
init_debug(1);
↦ Debugging level is set to  1
poly f=(x2+3y-2z)^2+xyz-(x-y3+x2*z3)^3;
poly g=morsesplit(f);
↦ Residual singularity f= -x3+3/2xy2+1/2x3y-1/16x2y2+3x2y3
g;
↦ -x3+3/2xy2+1/2x3y-1/16x2y2+3x2y3
```

D.5.1.9 quickclass

Procedure from library `classify.lib` (see Section D.5.1 [classify_lib], page 426).

Usage: quickclass(f); f=poly

Return: Normal form of f in Arnold's list

Remark: try to determine the normal form of f by invariants, mainly by computing the Hilbert function of the Milnor algebra, no coordinate change is needed (see also proc 'milnorcode').

Example:

```
LIB "classify.lib";
ring r=0,(x,y,z),ds;
poly f=(x2+3y-2z)^2+xyz-(x-y3+x2*z3)^3;
quickclass(f);
↦ Singularity R-equivalent to :  D[k]=D[4]
↦ normal form : z2+x2y+y3
↦ z2+x2y+y3
```

D.5.1.10 singularity

Procedure from library `classify.lib` (see Section D.5.1 [classify_lib], page 426).

Usage: singularity(t, l); t=string (name of singularity),
l=list of integers/polynomials (indices/parameters of singularity)

Compute: get the singularity named by type t from the database. list l is as follows:
l= k [,r [,s [,a [,b [,c [,d]..]: k,r,s=int a,b,c,d=poly.
The name of the dbm-database file is: NFlist.[dir,pag]. The file is found in the current directory. If it does not exist, please run the script MakeDBM first.

Return: Normal form and corank of the singularity named by type t and its index (indices) l.

Example:

```
LIB "classify.lib";
ring r=0,(x,y,z),(c,ds);
init_debug(0);
singularity("E[6k]",6);
↦ [1]:
↦    x3+xy13+y19
↦ [2]:
↦    2
singularity("T[k,r,s]", 3, 7, 5);
↦ [1]:
↦    x3+xyz+z5+y7
↦ [2]:
↦    3
poly f=y;
singularity("J[k,r]", 4, 0, 0, f);
↦ [1]:
↦    x3+x2y4+y13
↦ [2]:
↦    2
```

D.5.1.11 swap

Procedure from library `classify.lib` (see Section D.5.1 [classify_lib], page 426).

Usage: swap(a,b);

Return: b,a if b,a is the input (any type)

Example:

```
LIB "classify.lib";
swap("variable1","variable2");
↦ variable2 variable1
```

D.5.1.12 A_L

Procedure from library `classify.lib` (see Section D.5.1 [classify_lib], page 426).

Usage: A_L(f); f poly
A_L(s); s string, the name of the singularity

Compute: the normal form of f in Arnold's list of singularities in case 1, in case 2 nothing has to be computed.

Return: A_L(f): compute via 'milnorcode' the class of f and return the normal form of f found in the database.
A_L("name"): get the normal form from the database for the singularity given by its name.

Example:

```
LIB "classify.lib";
ring r=0,(a,b,c),ds;
poly f=A_L("E[13]");
f;
↦ c2+a3+ab5+b8
A_L(f);
↦ Singularity R-equivalent to :  E[6k+1]=E[13]
↦ normal form : c2+a3+ab5+b8
↦ c2+a3+ab5+b8
```

D.5.1.13 normalform

Procedure from library `classify.lib` (see Section D.5.1 [classify_lib], page 426).

Usage: normalform(s); s=string

Return: Arnold's normal form of singularity with name s

Example:

```
LIB "classify.lib";
ring r=0,(a,b,c),ds;
normalform("E[13]");
↦ c2+a3+ab5+b8
```

D.5.1.14 debug_log

Procedure from library `classify.lib` (see Section D.5.1 [classify_lib], page 426).

Usage: debug_log(level,li); level=int, li=comma separated "message" list

Compute: print "messages" if level>=@DeBug.
useful for user-defined trace messages.

Example:

```
LIB "classify.lib";
example init_debug;
↦ // proc init_debug from lib classify.lib
↦ EXAMPLE:
↦   init_debug();
↦   debug_log(1,"no trace information printed");
↦   init_debug(1);
↦ Debugging level change from  0  to  1
↦   debug_log(1,"some trace information");
↦ some trace information
↦   init_debug(2);
↦ Debugging level change from  1  to  2
↦   debug_log(2,"nice for debugging scripts");
↦ Debug:(2):  nice for debugging scripts
↦   init_debug(0);
↦ Debugging switched off.
↦
```

See also: Section D.5.1.5 [init_debug], page 429.

D.5.2 deform_lib

Library: deform.lib

Purpose: Miniversal Deformation of Singularities and Modules

Author: Bernd Martin, email: martin@math.tu-cottbus.de

Procedures:

D.5.2.1 versal

Procedure from library `deform.lib` (see Section D.5.2 [deform_lib], page 434).

Usage: versal(Fo[,d,any]); Fo=ideal, d=int, any=list

Compute: miniversal deformation of Fo up to degree d (default d=100),

Create: Rings (exported):
'my'Px = extending the basering Po by new variables given by "A,B,.." (deformation parameters), returns as basering; the new variables precede the old ones, the ordering is the product between "ls" and "ord(Po)"
'my'Qx = Px/Fo extending Qo=Po/Fo,
'my'So = the embedding-ring of the versal base space,
'my'Ox = Px/Js extending So/Js. (default my="")

Matrices (in Px, exported):
Js = giving the versal base space (obstructions),

Fs = giving the versal family of Fo,
Rs = giving the lifting of Ro=syz(Fo).

If d is defined (!=0), it computes up to degree d.
If 'any' is defined and any[1] is no string, interactive version.
Otherwise 'any' gives predefined strings: "my","param","order","out" ("my" prefix-string, "param" is a letter (e.g. "A") for the name of first parameter or (e.g. "A(") for index parameter variables, "order" ordering string for ring extension), "out" name of output-file).

Note: printlevel < 0 no output at all,
printlevel >=0,1,2,.. informs you, what is going on;
this proc uses 'execute'.

Example:

```
LIB "deform.lib";
int p          = printlevel;
printlevel     = 0;
ring r1        = 0,(x,y,z,u,v),ds;
matrix m[2][4] = x,y,z,u,y,z,u,v;
ideal Fo       = minor(m,2);
// cone over rational normal curve of degree 4
versal(Fo);
↦ // ready: T_1 and T_2
↦ // start computation in degree 2.
↦
↦ // 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;
// ___ Equations of miniversal base space ___:
Js;"";
↦ Js[1,1]=BD
↦ Js[1,2]=-AD+D2
↦ Js[1,3]=-CD
↦
// ___ Equations of miniversal total space ___:
Fs;"";
↦ Fs[1,1]=-u2+zv+Bu+Dv
↦ Fs[1,2]=-zu+yv-Au+Du
↦ Fs[1,3]=-yu+xv+Cu+Dz
↦ Fs[1,4]=z2-yu+Az+By
↦ Fs[1,5]=yz-xu+Bx-Cz
↦ Fs[1,6]=-y2+xz+Ax+Cy
↦
```

D.5.2.2 mod_versal

Procedure from library `deform.lib` (see Section D.5.2 [deform_lib], page 434).

Usage: mod_versal(Mo,I[,d,any]); I=ideal, M=module, d=int, any =list

Compute: miniversal deformation of coker(Mo) over Qo=Po/Io, Po=basering;

Create: Ringsr (exported):
'my'Px = extending the basering by new variables (deformation parameters), the new variables precede the old ones,
the ordering is the product between "my_ord"
and "ord(Po)"
'my'Qx = Px/Io extending Qo (returns as basering),
'my'Ox = Px/(Io+Js) ring of the versal deformation of coker(Ms),
'my'So = embedding-ring of the versal base space. (default 'my'="")

Matrices (in Qx, exported):
Js = giving the versal base space (obstructions),
Ms = giving the versal family of Mo,
Ls = giving the lifting of syzygies Lo=syz(Mo),

If d is defined (!=0), it computes up to degree d.
If 'any' is defined and any[1] is no string, interactive version.
Otherwise 'any' gives predefined strings:"my","param","order","out" ("my" prefix-string, "param" is a letter (e.g. "A") for the name of first parameter or (e.g. "A(") for index parameter variables, "ord" ordering string for ring extension), "out" name of output-file).

Note: printlevel < 0 no output at all,
printlevel >=0,1,2,.. informs you, what is going on,
this proc uses 'execute'.

Example:

```
LIB "deform.lib";
int p = printlevel;
printlevel = 1;
ring  Ro = 0,(x,y),wp(3,4);
ideal Io = x4+y3;
matrix Mo[2][2] = x2,y,-y2,x2;
mod_versal(Mo,Io);
↦ // vdim (Ext^2) = 4
↦ // vdim (Ext^1) = 4
↦ // ready: Ext1 and Ext2
↦ // Ext1 is quasi-homogeneous represented: 3,6,1,4
↦ // infinitesimal extension
↦ x2-Ax-B,   y+Cx+D,
↦ -y2+Cxy+Dy,x2+Ax+B
↦ // start deg = 2
↦ // start deg = 3
↦ // start deg = 4
↦ // start deg = 5
↦ // finished in degree
↦ 5
↦ // quasi-homogeneous weights of miniversal base
↦ 3,6,1,4
↦ // Result belongs to qring Qx
↦ // Equations of total space of miniversal deformation are in Js
↦ -2AB+A3+3CD2-BC3-3AC2D+A2C3,
```

```
↦ -B2+A2B+D3-3BC2D+ABC3
↦ // Matrix of the deformed module is Ms and lifted syzygies are Ls.
↦ // Make Qx the basering and list objects defined in Qx by typing:
↦    listvar(ring);setring Qx; show(Qx);listvar(ideal);listvar(matrix);
↦ // NOTE: rings Qx, Ox, So are still alive!
↦ // (use: 'kill_rings();' to remove them)
printlevel = p;
if(system("with","Namespaces")) {
if(nameof(Current) == "Ring" ) {
kill Top::Px,Top::Qx,Top::So;
} else {
kill Ring::Px,Ring::So;
}
}
kill Px,Qx,So;
```

D.5.2.3 lift_kbase

Procedure from library `deform.lib` (see Section D.5.2 [deform_lib], page 434).

Usage: lift_kbase(N,M); N,M=poly/ideal/vector/module

Return: matrix A, coefficient matrix expressing N as linear combination of k-basis of M. Let the k-basis have k elements and size(N)=c columns. Then A satisfies:
matrix(reduce(N,std(M)),k,c) = matrix(kbase(std(M)))*A

Assume: dim(M)=0 and the monomial ordering is a well ordering or the last block of the ordering is c or C

Example:

```
LIB "deform.lib";
ring R=0,(x,y),ds;
module M=[x2,xy],[y2,xy],[0,xx],[0,yy];
module N=[x3+xy,x],[x,x+y2];
print(M);
↦ x2,y2,0, 0,
↦ xy,xy,x2,y2
module kb=kbase(std(M));
print(kb);
↦ y2,xy,y,x,1,0,0,0,
↦ 0, 0, 0,0,0,y,x,1
print(N);
↦ xy+x3,x,
↦ x,    x+y2
matrix A=lift_kbase(N,M);
print(A);
↦ 0,0,
↦ 1,0,
↦ 0,0,
↦ 0,1,
↦ 0,0,
↦ 0,0,
↦ 1,1,
↦ 0,0
matrix(reduce(N,std(M)),nrows(kb),ncols(A)) - matrix(kbase(std(M)))*A;
```

```
↦ _[1,1]=0
↦ _[1,2]=0
↦ _[2,1]=0
↦ _[2,2]=0
```

D.5.2.4 lift_rel_kb

Procedure from library `deform.lib` (see Section D.5.2 [deform_lib], page 434).

Usage: lift_rel_kb(N,M[,kbaseM,p]);

Assume: [p a monomial] or the product of all variables
N, M modules of same rank, M depending only on variables not in p and vdim(M) is finite in this ring,
[kbaseM the kbase of M in the subring given by variables not in p]
warning: these assumptions are not checked by the procedure

Return: matrix A, whose j-th columns present the coeff's of N[j] in kbaseM, i.e. kbaseM*A = reduce(N,std(M))

Example:

```
LIB "deform.lib";
ring r=0,(A,B,x,y),dp;
module M      = [x2,xy],[xy,y3],[y2],[0,x];
module kbaseM = [1],[x],[xy],[y],[0,1],[0,y],[0,y2];
poly f=xy;
module N = [AB,BBy],[A3xy+x4,AB*(1+y2)];
matrix A = lift_rel_kb(N,M,kbaseM,f);
print(A);
↦ AB,0,
↦ 0, 0,
↦ 0, A3,
↦ 0, 0,
↦ 0, AB,
↦ B2,0,
↦ 0, AB
"TEST:";
↦ TEST:
print(matrix(kbaseM)*A-matrix(reduce(N,std(M))));
↦ 0,0,
↦ 0,0
```

D.5.2.5 kill_rings

Procedure from library `deform.lib` (see Section D.5.2 [deform_lib], page 434).

Usage: kill_rings([string]);

Return: nothing, but kills exported rings generated by procedures 'versal' and 'mod_versal' with optional prefix 'string'

D.5.3 equising_lib

Library: equising.lib

Purpose: Equisingularity Stratum of a Family of Plane Curves

Author: Andrea Mindnich, mindnich@mathematik.uni-kl.de

Procedures:

D.5.3.1 esStratum

Procedure from library `equising.lib` (see Section D.5.3 [equising_lib], page 438).

Usage: esStratum(F[,m]); F poly, m int

Assume: F defines a deformation of an irreducible bivariate polynomial f and the characteristic of the basering does not divide mult(f).
If nv is the number of variables of the basering, then the first nv-2 variables are the deformation parameters.
If the basering is a qring, ideal(basering) must only depend on the deformation parameters.

Return: list l of an ideal and an integer, where
l[1] is the ideal in the deformation parameters, defining the ES-stratum of F,
l[2]=1 if some error has occured, l[2]=0 otherwise.

Note: If m is given, the computation stops after m steps of the iteration.
printlevel > 0 displays comments and pauses after intermediate computations (default: printlevel=0)
This procedure uses `execute` or calls a procedure using `execute`.

Example:

```
LIB "equising.lib";
ring r = 11,(a,b,c,d,e,f,g,x,y),ds;
poly F = (x2+2xy+y2+x5)+ax+by+cx2+dxy+ey2+fx3+gx4;
esStratum(F);
↦ [1]:
↦    _[1]=g
↦    _[2]=f
↦    _[3]=b
↦    _[4]=a
↦    _[5]=-4c+4d-4e+d2-4ce
↦ [2]:
↦    0
esStratum(F,2);
↦ [1]:
↦    _[1]=g
↦    _[2]=f
↦    _[3]=b
↦    _[4]=a
↦    _[5]=-4c+4d-4e+d2-4ce
↦ [2]:
↦    0
ideal I = f-fa,e+b;
qring q = std(I);
poly F = imap(r,F);
esStratum(F);
↦ [1]:
↦    _[1]=a
↦    _[2]=-e
```

```
↦     _[3]=c-d-3d2
↦     _[4]=g
↦  [2]:
↦     0
```

D.5.3.2 isEquising

Procedure from library `equising.lib` (see Section D.5.3 [equising_lib], page 438).

Usage: isEquising(F[,m]); F poly, m int

Assume: F defines a deformation of an irreducible bivariate polynomial f and the characteristic of the basering does not divide mult(f).
If nv is the number of variables of the basering, then the first nv-2 variables are the deformation parameters.
If the basering is a qring, ideal(basering) must only depend on the deformation parameters.

Return: list l of two integers, where
l[1]=1 if F is an equisingular deformation, l[1]=0 otherwise.
l[2]=1 if some error has occured, l[2]=0 otherwise.

Note: If m is given, the computation stops after m steps of the iteration.
This procedure uses `execute` or calls a procedure using `execute`.

Example:

```
LIB "equising.lib";
ring r = 11,(a,b,x,y),ds;
poly F = (x2+2xy+y2+x5)+ay3+bx5;
isEquising(F);
↦  [1]:
↦     0
↦  [2]:
↦     0
isEquising(F,1);
↦  [1]:
↦     1
↦  [2]:
↦     0
ideal I = ideal(a);
qring q = std(I);
poly F = imap(r,F);
isEquising(F);
↦  [1]:
↦     1
↦  [2]:
↦     0
```

D.5.4 gaussman_lib

Library: gaussman.lib

Purpose: Algorithmic Gauss-Manin Connection

Author: Mathias Schulze, email: mschulze@mathematik.uni-kl.de

Overview: A library to compute Hodge-theoretic invariants
of isolated hypersurface singularities

Procedures: **See also:** Section D.5.6 [mondromy_lib], page 473; Section D.5.10 [spectrum_lib], page 495.

D.5.4.1 gmsring

Procedure from library `gaussman.lib` (see Section D.5.4 [gaussman_lib], page 440).

Usage: gmsring(t,s); poly t, string s

Assume: characteristic 0; local degree ordering;
isolated critical point 0 of t

Return:

ring G; Gauss-Manin system of t with variable s
poly gmspoly=t;
ideal gmsjacob; Jacobian ideal of t
ideal gmsstd; standard basis of Jacobian ideal
matrix gmsmatrix; matrix(gmsjacob)*gmsmatrix==matrix(gmsstd)
ideal gmsbasis; monomial vector space basis of Jacobian algebra
int gmsmaxdeg; maximal weight of variables

Note: gmsbasis is a C[[s]]-basis of H" and [t,s]=s^2

Example:

```
LIB "gaussman.lib";
ring R=0,(x,y),ds;
poly t=x5+x2y2+y5;
def G=gmsring(t,"s");
setring(G);
gmspoly;
↦ x2y2+x5+y5
print(gmsjacob);
↦ 2xy2+5x4,
↦ 2x2y+5y4
print(gmsstd);
↦ 2x2y+5y4,
↦ 2xy2+5x4,
↦ 5x5-5y5,
↦ 10y6+25x3y4
print(gmsmatrix);
↦ 0,1,x, -2xy,
↦ 1,0,-y,2y2+5x3
print(gmsbasis);
↦ y5,
↦ y4,
↦ y3,
↦ y2,
↦ xy,
↦ y,
↦ x4,
↦ x3,
↦ x2,
```

```
↦ x,
↦ 1
gmsmaxdeg;
↦ 1
```

D.5.4.2 gmsnf

Procedure from library `gaussman.lib` (see Section D.5.4 [gaussman_lib], page 440).

Usage: gmsnf(p,K); poly p, int K

Assume: basering returned by gmsring

Return:

list nf;
  ideal nf[1]; projection of p to <gmsbasis>C[[s]] mod s^(K+1)
  ideal nf[2]; p==nf[1]+nf[2]

Note: computation can be continued by setting p=nf[2]

Example:

```
LIB "gaussman.lib";
ring R=0,(x,y),ds;
poly t=x5+x2y2+y5;
def G=gmsring(t,"s");
setring(G);
list l0=gmsnf(gmspoly,0);
print(l0[1]);
↦ -1/2y5
list l1=gmsnf(gmspoly,1);
print(l1[1]);
↦ -1/2y5+1/2s
list l=gmsnf(l0[2],1);
print(l[1]);
↦ 1/2s
```

D.5.4.3 gmscoeffs

Procedure from library `gaussman.lib` (see Section D.5.4 [gaussman_lib], page 440).

Usage: gmscoeffs(p,K); poly p, int K

Assume: basering constructed by gmsring

Return:

list l;
  matrix l[1]; C[[s]]-basis representation of p mod s^(K+1)
  ideal l[2]; p==matrix(gmsbasis)*l[1]+l[2]

Note: computation can be continued by setting p=l[2]

Example:

```
LIB "gaussman.lib";
ring R=0,(x,y),ds;
poly t=x5+x2y2+y5;
def G=gmsring(t,"s");
setring(G);
```

```
list l0=gmscoeffs(gmspoly,0);
print(l0[1]);
↦ -1/2,
↦ 0,
↦ 0,
↦ 0,
↦ 0,
↦ 0,
↦ 0,
↦ 0,
↦ 0,
↦ 0,
↦ 0
list l1=gmscoeffs(gmspoly,1);
print(l1[1]);
↦ -1/2,
↦ 0,
↦ 0,
↦ 0,
↦ 0,
↦ 0,
↦ 0,
↦ 0,
↦ 0,
↦ 0,
↦ 1/2s
list l=gmscoeffs(l0[2],1);
print(l[1]);
↦ 0,
↦ 0,
↦ 0,
↦ 0,
↦ 0,
↦ 0,
↦ 0,
↦ 0,
↦ 0,
↦ 0,
↦ 1/2s
```

D.5.4.4 monodromy

Procedure from library `gaussman.lib` (see Section D.5.4 [gaussman_lib], page 440).

Usage: monodromy(t); poly t

Assume: characteristic 0; local degree ordering;
isolated critical point 0 of t

Return:

list l; Jordan data jordan(M) of monodromy matrix exp(-2*pi*i*M)
  ideal l[1];
    number l[1][i]; eigenvalue of i-th Jordan block of M
  intvec l[2];

int l[2][i]; size of i-th Jordan block of M
intvec l[3];
int l[3][i]; multiplicity of i-th Jordan block of M

Example:

```
LIB "gaussman.lib";
ring R=0,(x,y),ds;
poly t=x5+x2y2+y5;
monodromy(t);
↦ [1]:
↦    _[1]=1/2
↦    _[2]=7/10
↦    _[3]=9/10
↦    _[4]=1
↦    _[5]=11/10
↦    _[6]=13/10
↦ [2]:
↦    2,1,1,1,1,1
↦ [3]:
↦    1,2,2,1,2,2
```

See also: Section D.3.2 [linalg_lib], page 353; Section D.5.6 [mondromy_lib], page 473.

D.5.4.5 spectrum

Procedure from library `gaussman.lib` (see Section D.5.4 [gaussman_lib], page 440).

Usage: spectrum(t); poly t

Assume: characteristic 0; local degree ordering;
isolated critical point 0 of t

Return:

list sp; singularity spectrum of t
ideal sp[1];
number sp[1][i]; i-th spectral number
intvec sp[2];
int sp[2][i]; multiplicity of i-th spectral number

Example:

```
LIB "gaussman.lib";
ring R=0,(x,y),ds;
poly t=x5+x2y2+y5;
spprint(spectrum(t));
↦ (-1/2,1),(-3/10,2),(-1/10,2),(0,1),(1/10,2),(3/10,2),(1/2,1)
```

See also: Section D.5.10 [spectrum_lib], page 495.

D.5.4.6 sppairs

Procedure from library `gaussman.lib` (see Section D.5.4 [gaussman_lib], page 440).

Usage: sppairs(t); poly t

Assume: characteristic 0; local degree ordering;
isolated critical point 0 of t

Return:

list spp; spectral pairs of t
  ideal spp[1];
    number spp[1][i]; V-filtration index of i-th spectral pair
  intvec spp[2];
    int spp[2][i]; weight filtration index of i-th spectral pair
  intvec spp[3];
    int spp[3][i]; multiplicity of i-th spectral pair

Example:

```
LIB "gaussman.lib";
ring R=0,(x,y),ds;
poly t=x5+x2y2+y5;
sppprint(sppairs(t));
↦ ((-1/2,2),1),((-3/10,1),2),((-1/10,1),2),((0,1),1),((1/10,1),2),((3/10,1)\
   ,2),((1/2,0),1)
```

See also: Section D.5.10 [spectrum_lib], page 495.

D.5.4.7 spnf

Procedure from library `gaussman.lib` (see Section D.5.4 [gaussman_lib], page 440).

Assume: ncols(a)==size(m)==size(V); typeof(V[i])=="int"

Return: order (a[i][,V[i]]) with multiplicity m[i] lexicographically

D.5.4.8 sppnf

Procedure from library `gaussman.lib` (see Section D.5.4 [gaussman_lib], page 440).

Assume: ncols(e)=size(w)=size(m)=size(V); typeof(V[i])=="module"

Return: order (a[i][,w[i]][,V[i]]) with multiplicity m[i] lexicographically

D.5.4.9 vfilt

Procedure from library `gaussman.lib` (see Section D.5.4 [gaussman_lib], page 440).

Usage: vfilt(t); poly t

Assume: characteristic 0; local degree ordering;
isolated critical point 0 of t

Return:

list v; V-filtration on H”/s*H”
  ideal v[1];
    number v[1][i]; V-filtration index of i-th spectral number
  intvec v[2];
    int v[2][i]; multiplicity of i-th spectral number
  list v[3];
    module v[3][i]; vector space of i-th graded part in terms of v[4]
  ideal v[4]; monomial vector space basis of H”/s*H”
  ideal v[5]; standard basis of Jacobian ideal

Example:

```
LIB "gaussman.lib";
ring R=0,(x,y),ds;
poly t=x5+x2y2+y5;
vfilt(t);
↦ [1]:
↦    _[1]=-1/2
↦    _[2]=-3/10
↦    _[3]=-1/10
↦    _[4]=0
↦    _[5]=1/10
↦    _[6]=3/10
↦    _[7]=1/2
↦ [2]:
↦    1,2,2,1,2,2,1
↦ [3]:
↦    [1]:
↦       _[1]=gen(11)
↦    [2]:
↦       _[1]=gen(10)
↦       _[2]=gen(6)
↦    [3]:
↦       _[1]=gen(9)
↦       _[2]=gen(4)
↦    [4]:
↦       _[1]=gen(5)
↦    [5]:
↦       _[1]=gen(3)
↦       _[2]=gen(8)
↦    [6]:
↦       _[1]=gen(2)
↦       _[2]=gen(7)
↦    [7]:
↦       _[1]=gen(1)
↦ [4]:
↦    _[1]=y5
↦    _[2]=y4
↦    _[3]=y3
↦    _[4]=y2
↦    _[5]=xy
↦    _[6]=y
↦    _[7]=x4
↦    _[8]=x3
↦    _[9]=x2
↦    _[10]=x
↦    _[11]=1
↦ [5]:
↦    _[1]=2x2y+5y4
↦    _[2]=2xy2+5x4
↦    _[3]=5x5-5y5
↦    _[4]=10y6+25x3y4
```

See also: Section D.5.10 [spectrum_lib], page 495.

D.5.4.10 vwfilt

Procedure from library `gaussman.lib` (see Section D.5.4 [gaussman_lib], page 440).

Usage: vwfilt(t); poly t

Assume: characteristic 0; local degree ordering;
isolated critical point 0 of t

Return:

list vw; weighted V-filtration on H"/s*H"
  ideal vw[1];
    number vw[1][i]; V-filtration index of i-th spectral pair
  intvec vw[2];
    int vw[2][i]; weight filtration index of i-th spectral pair
  intvec vw[3];
    int vw[3][i]; multiplicity of i-th spectral pair
  list vw[4];
    module vw[4][i]; vector space of i-th graded part in terms of vw[5]
  ideal vw[5]; monomial vector space basis of H"/s*H"
  ideal vw[6]; standard basis of Jacobian ideal

Example:

```
LIB "gaussman.lib";
ring R=0,(x,y),ds;
poly t=x5+x2y2+y5;
vwfilt(t);
↦ [1]:
↦    _[1]=-1/2
↦    _[2]=-3/10
↦    _[3]=-1/10
↦    _[4]=0
↦    _[5]=1/10
↦    _[6]=3/10
↦    _[7]=1/2
↦ [2]:
↦    2,1,1,1,1,1,0
↦ [3]:
↦    1,2,2,1,2,2,1
↦ [4]:
↦    [1]:
↦       _[1]=gen(11)
↦    [2]:
↦       _[1]=gen(10)
↦       _[2]=gen(6)
↦    [3]:
↦       _[1]=gen(9)
↦       _[2]=gen(4)
↦    [4]:
↦       _[1]=gen(5)
↦    [5]:
↦       _[1]=gen(3)
↦       _[2]=gen(8)
↦    [6]:
↦       _[1]=gen(2)
```

```
↦        _[2]=gen(7)
↦     [7]:
↦        _[1]=gen(1)
↦  [5]:
↦     _[1]=y5
↦     _[2]=y4
↦     _[3]=y3
↦     _[4]=y2
↦     _[5]=xy
↦     _[6]=y
↦     _[7]=x4
↦     _[8]=x3
↦     _[9]=x2
↦     _[10]=x
↦     _[11]=1
↦  [6]:
↦     _[1]=2x2y+5y4
↦     _[2]=2xy2+5x4
↦     _[3]=5x5-5y5
↦     _[4]=10y6+25x3y4
```

See also: Section D.5.10 [spectrum_lib], page 495.

D.5.4.11 tmatrix

Procedure from library `gaussman.lib` (see Section D.5.4 [gaussman_lib], page 440).

Usage: tmatrix(t); poly t

Assume: characteristic 0; local degree ordering;
isolated critical point 0 of t

Return:

list A; C[[s]]-matrix A[1]+s*A[2] of t on H”
matrix A[1];
matrix A[2];

Example:

```
LIB "gaussman.lib";
ring R=0,(x,y),ds;
poly t=x5+x2y2+y5;
list A=tmatrix(t);
print(A[1]);
↦ 0,0,0,0,0,0,0,0,0,0,0,
↦ 0,0,0,0,0,0,0,0,0,0,0,
↦ 0,0,0,0,0,0,0,0,0,0,0,
↦ 0,0,0,0,0,0,0,0,0,0,0,
↦ 0,0,0,0,0,0,0,0,0,0,0,
↦ 0,0,0,0,0,0,0,0,0,0,0,
↦ 0,0,0,0,0,0,0,0,0,0,0,
↦ 0,0,0,0,0,0,0,0,0,0,0,
↦ 0,0,0,0,0,0,0,0,0,0,0,
↦ 0,0,0,0,0,0,0,0,0,0,0,
↦ 1,0,0,0,0,0,0,0,0,0,0
print(A[2]);
```

```
↦ 1/2,0,   0,   0,   0,   0,0,    0,    0,    0,    0,
↦ 0,  7/10,0,   0,   0,   0,0,    0,    0,    0,    0,
↦ 0,  0,   7/10,0,   0,   0,0,    0,    0,    0,    0,
↦ 0,  0,   0,   9/10,0,   0,0,    0,    0,    0,    0,
↦ 0,  0,   0,   0,   9/10,0,0,    0,    0,    0,    0,
↦ 0,  0,   0,   0,   0,   1,0,    0,    0,    0,    0,
↦ 0,  0,   0,   0,   0,   0,11/10,0,    0,    0,    0,
↦ 0,  0,   0,   0,   0,   0,0,    11/10,0,    0,    0,
↦ 0,  0,   0,   0,   0,   0,0,    0,    13/10,0,    0,
↦ 0,  0,   0,   0,   0,   0,0,    0,    0,    13/10,0,
↦ 0,  0,   0,   0,   0,   0,0,    0,    0,    0,    3/2
```

D.5.4.12 endvfilt

Procedure from library `gaussman.lib` (see Section D.5.4 [gaussman_lib], page 440).

Usage: endvfilt(v); list v

Assume: v returned by vfilt

Return:

list ev; V-filtration on Jacobian algebra
  ideal ev[1];
    number ev[1][i]; i-th V-filtration index
  intvec ev[2];
    int ev[2][i]; i-th multiplicity
  list ev[3];
    module ev[3][i]; vector space of i-th graded part in terms of ev[4]
  ideal ev[4]; monomial vector space basis of Jacobian algebra
  ideal ev[5]; standard basis of Jacobian ideal

Example:

```
LIB "gaussman.lib";
ring R=0,(x,y),ds;
poly t=x5+x2y2+y5;
endvfilt(vfilt(t));
↦ [1]:
↦    _[1]=0
↦    _[2]=1/5
↦    _[3]=2/5
↦    _[4]=1/2
↦    _[5]=3/5
↦    _[6]=4/5
↦    _[7]=1
↦ [2]:
↦    1,2,2,1,2,2,1
↦ [3]:
↦    [1]:
↦       _[1]=gen(11)
↦    [2]:
↦       _[1]=gen(10)
↦       _[2]=gen(6)
↦    [3]:
↦       _[1]=gen(9)
↦       _[2]=gen(4)
```

```
↦     [4]:
↦        _[1]=gen(5)
↦     [5]:
↦        _[1]=gen(8)
↦        _[2]=gen(3)
↦     [6]:
↦        _[1]=gen(7)
↦        _[2]=gen(2)
↦     [7]:
↦        _[1]=gen(1)
↦  [4]:
↦     _[1]=y5
↦     _[2]=y4
↦     _[3]=y3
↦     _[4]=y2
↦     _[5]=xy
↦     _[6]=y
↦     _[7]=x4
↦     _[8]=x3
↦     _[9]=x2
↦     _[10]=x
↦     _[11]=1
↦  [5]:
↦     _[1]=2x2y+5y4
↦     _[2]=2xy2+5x4
↦     _[3]=5x5-5y5
↦     _[4]=10y6+25x3y4
```

D.5.4.13 spprint

Procedure from library `gaussman.lib` (see Section D.5.4 [gaussman_lib], page 440).

Usage: spprint(sp); list sp

Return: string s; spectrum sp

Example:

```
LIB "gaussman.lib";
ring R=0,(x,y),ds;
list sp=list(ideal(-1/2,-3/10,-1/10,0,1/10,3/10,1/2),intvec(1,2,2,1,2,2,1));
spprint(sp);
↦ (-1/2,1),(-3/10,2),(-1/10,2),(0,1),(1/10,2),(3/10,2),(1/2,1)
```

D.5.4.14 sppprint

Procedure from library `gaussman.lib` (see Section D.5.4 [gaussman_lib], page 440).

Usage: sppprint(spp); list spp

Return: string s; spectral pairs spp

Example:

```
LIB "gaussman.lib";
ring R=0,(x,y),ds;
list spp=list(ideal(-1/2,-3/10,-1/10,0,1/10,3/10,1/2),intvec(2,1,1,1,1,1,0),intvec(1,2,2,1,2,2,1));
sppprint(spp);
```

```
↦ ((-1/2,2),1),((-3/10,1),2),((-1/10,1),2),((0,1),1),((1/10,1),2),((3/10,1)\
   ,2),((1/2,0),1)
```

D.5.4.15 spadd

Procedure from library `gaussman.lib` (see Section D.5.4 [gaussman_lib], page 440).

Usage: spadd(sp1,sp2); list sp1, list sp2

Return: list sp; sum of spectra sp1 and sp2

Example:

```
LIB "gaussman.lib";
ring R=0,(x,y),ds;
list sp1=list(ideal(-1/2,-3/10,-1/10,0,1/10,3/10,1/2),intvec(1,2,2,1,2,2,1));
spprint(sp1);
↦ (-1/2,1),(-3/10,2),(-1/10,2),(0,1),(1/10,2),(3/10,2),(1/2,1)
list sp2=list(ideal(-1/6,1/6),intvec(1,1));
spprint(sp2);
↦ (-1/6,1),(1/6,1)
spprint(spadd(sp1,sp2));
↦ (-1/2,1),(-3/10,2),(-1/6,1),(-1/10,2),(0,1),(1/10,2),(1/6,1),(3/10,2),(1/\
   2,1)
```

D.5.4.16 spsub

Procedure from library `gaussman.lib` (see Section D.5.4 [gaussman_lib], page 440).

Usage: spsub(sp1,sp2); list sp1, list sp2

Return: list sp; difference of spectra sp1 and sp2

Example:

```
LIB "gaussman.lib";
ring R=0,(x,y),ds;
list sp1=list(ideal(-1/2,-3/10,-1/10,0,1/10,3/10,1/2),intvec(1,2,2,1,2,2,1));
spprint(sp1);
↦ (-1/2,1),(-3/10,2),(-1/10,2),(0,1),(1/10,2),(3/10,2),(1/2,1)
list sp2=list(ideal(-1/6,1/6),intvec(1,1));
spprint(sp2);
↦ (-1/6,1),(1/6,1)
spprint(spsub(sp1,sp2));
↦ (-1/2,1),(-3/10,2),(-1/6,-1),(-1/10,2),(0,1),(1/10,2),(1/6,-1),(3/10,2),(\
   1/2,1)
```

D.5.4.17 spmul

Procedure from library `gaussman.lib` (see Section D.5.4 [gaussman_lib], page 440).

Usage: spmul(sp0,k); list sp0, int[vec] k

Return: list sp; linear combination of spectra sp0 with coefficients k

Example:

```
LIB "gaussman.lib";
ring R=0,(x,y),ds;
list sp=list(ideal(-1/2,-3/10,-1/10,0,1/10,3/10,1/2),intvec(1,2,2,1,2,2,1));
```

```
spprint(sp);
↦ (-1/2,1),(-3/10,2),(-1/10,2),(0,1),(1/10,2),(3/10,2),(1/2,1)
spprint(spmul(sp,2));
↦ (-1/2,2),(-3/10,4),(-1/10,4),(0,2),(1/10,4),(3/10,4),(1/2,2)
list sp1=list(ideal(-1/6,1/6),intvec(1,1));
spprint(sp1);
↦ (-1/6,1),(1/6,1)
list sp2=list(ideal(-1/3,0,1/3),intvec(1,2,1));
spprint(sp2);
↦ (-1/3,1),(0,2),(1/3,1)
spprint(spmul(list(sp1,sp2),intvec(1,2)));
↦ (-1/3,2),(-1/6,1),(0,4),(1/6,1),(1/3,2)
```

D.5.4.18 spissemicont

Procedure from library `gaussman.lib` (see Section D.5.4 [gaussman_lib], page 440).

Usage: spissemicont(sp[,1]); list sp, int opt

Return:

int k=
1; if sum of sp is positive on all intervals [a,a+1) [and (a,a+1)]
0; if sum of sp is negative on some interval [a,a+1) [or (a,a+1)]

Example:

```
LIB "gaussman.lib";
ring R=0,(x,y),ds;
list sp1=list(ideal(-1/2,-3/10,-1/10,0,1/10,3/10,1/2),intvec(1,2,2,1,2,2,1));
spprint(sp1);
↦ (-1/2,1),(-3/10,2),(-1/10,2),(0,1),(1/10,2),(3/10,2),(1/2,1)
list sp2=list(ideal(-1/6,1/6),intvec(1,1));
spprint(sp2);
↦ (-1/6,1),(1/6,1)
spissemicont(spsub(sp1,spmul(sp2,5)));
↦ 1
spissemicont(spsub(sp1,spmul(sp2,5)),1);
↦ 0
spissemicont(spsub(sp1,spmul(sp2,6)));
↦ 0
```

D.5.4.19 spsemicont

Procedure from library `gaussman.lib` (see Section D.5.4 [gaussman_lib], page 440).

Usage: spsemicont(sp0,sp,k[,1]); list sp0, list sp

Return:

list l;
intvec l[i]; if the spectra sp0 occur with multiplicities k
in a deformation of a [quasihomogeneous] singularity
with spectrum sp then k<=l[i]

Example:

```
LIB "gaussman.lib";
ring R=0,(x,y),ds;
```

```
list sp0=list(ideal(-1/2,-3/10,-1/10,0,1/10,3/10,1/2),intvec(1,2,2,1,2,2,1));
spprint(sp0);
↦ (-1/2,1),(-3/10,2),(-1/10,2),(0,1),(1/10,2),(3/10,2),(1/2,1)
list sp1=list(ideal(-1/6,1/6),intvec(1,1));
spprint(sp1);
↦ (-1/6,1),(1/6,1)
list sp2=list(ideal(-1/3,0,1/3),intvec(1,2,1));
spprint(sp2);
↦ (-1/3,1),(0,2),(1/3,1)
list sp=sp1,sp2;
list l=spsemicont(sp0,sp);
l;
↦ [1]:
↦    5
↦ [2]:
↦    3,1
↦ [3]:
↦    1,2
spissemicont(spsub(sp0,spmul(sp,l[1])));
↦ 1
spissemicont(spsub(sp0,spmul(sp,l[1]-1)));
↦ 1
spissemicont(spsub(sp0,spmul(sp,l[1]+1)));
↦ 0
```

D.5.4.20 spmilnor

Procedure from library **gaussman.lib** (see Section D.5.4 [gaussman_lib], page 440).

Usage: spmilnor(sp); list sp

Return: int mu; Milnor number of spectrum sp

Example:

```
LIB "gaussman.lib";
ring R=0,(x,y),ds;
list sp=list(ideal(-1/2,-3/10,-1/10,0,1/10,3/10,1/2),intvec(1,2,2,1,2,2,1));
spprint(sp);
↦ (-1/2,1),(-3/10,2),(-1/10,2),(0,1),(1/10,2),(3/10,2),(1/2,1)
spmilnor(sp);
↦ 11
```

D.5.4.21 spgeomgenus

Procedure from library **gaussman.lib** (see Section D.5.4 [gaussman_lib], page 440).

Usage: spgeomgenus(sp); list sp

Return: int g; geometrical genus of spectrum sp

Example:

```
LIB "gaussman.lib";
ring R=0,(x,y),ds;
list sp=list(ideal(-1/2,-3/10,-1/10,0,1/10,3/10,1/2),intvec(1,2,2,1,2,2,1));
spprint(sp);
↦ (-1/2,1),(-3/10,2),(-1/10,2),(0,1),(1/10,2),(3/10,2),(1/2,1)
```

```
spgeomgenus(sp);
↦ 6
```

D.5.4.22 spgamma

Procedure from library `gaussman.lib` (see Section D.5.4 [gaussman_lib], page 440).

Usage: spgamma(sp); list sp

Return: number gamma; gamma invariant of spectrum sp

Example:

```
LIB "gaussman.lib";
ring R=0,(x,y),ds;
list sp=list(ideal(-1/2,-3/10,-1/10,0,1/10,3/10,1/2),intvec(1,2,2,1,2,2,1));
spprint(sp);
↦ (-1/2,1),(-3/10,2),(-1/10,2),(0,1),(1/10,2),(3/10,2),(1/2,1)
spgamma(sp);
↦ 1/240
```

D.5.5 hnoether_lib

Library: hnoether.lib

Purpose: Hamburger-Noether (Puiseux) Development

Author: Martin Lamm, lamm@mathematik.uni-kl.de

Overview: A library for computing the Hamburger-Noether, resp. Puiseux, development of a plane curve singularity following [Campillo, A.: Algebroid curves in positive characteristic, Springer LNM 813 (1980)].
The library contains also procedures for computing the (topological) numerical invariants of plane curve singularities.

Main procedures: Auxiliary procedures:

D.5.5.1 HNdevelop

Procedure from library `hnoether.lib` (see Section D.5.5 [hnoether_lib], page 454).

Usage: HNdevelop(f); f poly

Assume: f is a bivariate polynomial (in the first 2 ring variables)

Create: ring with name `HNEring`, variables `x,y` and ordering `ls` over a field extension of the current basering's ground field.
Since the Hamburger-Noether development usually does not exist in the originally given basering, `HNdevelop` always defines `HNEring` and CHANGES to it. The field extension is chosen minimally.

Return: list `L` of lists `L[i]` (corresponding to the output of `develop(f[i])`, f[i] a branch of f, but the last entry being omitted).

`L[i][1]`; matrix:
Each row contains the coefficients of the corresponding line of the Hamburger-Noether expansion (HNE) for f[i]. The end of the line is marked in the matrix by the first ring variable (usually x).

`L[i][2]`; intvec:
indicating the length of lines of the HNE

`L[i][3]`; int:
0 if the 1st ring variable was transversal (with respect to f[i]),
1 if the variables were changed at the beginning of the computation,
-1 if an error has occurred.

`L[i][4]`; poly:
the transformed polynomial of f[i] to make it possible to extend the Hamburger-Noether development a posteriori without having to do all the previous calculation once again (0 if not needed)

Note: `HNdevelop` decides which procedure (`develop` or `reddevelop`) applies best to the given problem and calls it.
If f is known to be irreducible as a power series, `develop(f)` should be chosen instead to avoid the change of basering.
If `printlevel>=0` comments are displayed (default is `printlevel=0`).

Example:

```
LIB "hnoether.lib";
// -------- prepare for example ---------
if (nameof(basering)=="HNEring") {
def rettering=HNEring;
kill HNEring;
}
// ------ the example starts here -------
ring r=0,(x,y),dp;
list hne=HNdevelop(x4-y6);
↦ finite HNE found
↦ finite HNE found
↦ // result: 2 branch(es) successfully computed,
↦ //         basering has changed to HNEring
nameof(basering);
↦ HNEring
size(hne);           // number of branches
↦ 2
print(hne[1][1]);    // HN-matrix of 1st branch
↦ 0,x,0,
↦ 0,1,x
param(hne[1]);       // parametrization of 1st branch
↦ _[1]=x3
↦ _[2]=x2
param(hne[2]);       // parametrization of 2nd branch
↦ _[1]=-x3
↦ _[2]=-x2
kill HNEring,r;
// --- restore HNEring if previously defined ---
if (defined(rettering)) {
setring rettering;
def HNEring=rettering;
export HNEring;
}
```

See also: Section D.5.5.2 [develop], page 456; Section D.5.5.7 [displayHNE], page 463; Section D.5.5.4 [essdevelop], page 459; Section D.5.5.5 [extdevelop], page 460; Section D.5.5.6 [param], page 462; Section D.5.5.3 [reddevelop], page 457.

D.5.5.2 develop

Procedure from library `hnoether.lib` (see Section D.5.5 [hnoether_lib], page 454).

Usage: develop(f [,n]); f poly, n int

Assume: f is a bivariate polynomial (in the first 2 ring variables) and irreducible as power series (for reducible f use `reddevelop`).

Return: list `L` with:

`L[1]`; matrix:
: Each row contains the coefficients of the corresponding line of the Hamburger-Noether expansion (HNE). The end of the line is marked in the matrix by the first ring variable (usually x).

`L[2]`; intvec:
: indicating the length of lines of the HNE

`L[3]`; int: 0 if the 1st ring variable was transversal (with respect to f),
1 if the variables were changed at the beginning of the computation,
-1 if an error has occurred.

`L[4]`; poly:
: the transformed polynomial of f to make it possible to extend the Hamburger-Noether development a posteriori without having to do all the previous calculation once again (0 if not needed)

`L[5]`; int: 1 if the curve has exactly one branch (i.e., is irreducible),
0 else (i.e., the curve has more than one HNE, or f is not valid).

Display: The (non zero) elements of the HNE (if not called by another proc).

Note: The optional parameter `n` affects only the computation of the LAST line of the HNE. If it is given, the HN-matrix `L[1]` will have at least `n` columns.
Otherwise, the number of columns will be chosen minimal such that the matrix contains all necessary information (i.e., all lines of the HNE but the last (which is in general infinite) have place).
If `n` is negative, the algorithm is stopped as soon as the computed information is sufficient for `invariants(L)`, but the HN-matrix `L[1]` may still contain undetermined elements, which are marked with the 2nd variable (of the basering).
For time critical computations it is recommended to use `ring ...,(x,y),ls` as basering - it increases the algorithm's speed.
If `printlevel>=0` comments are displayed (default is `printlevel=0`).

Example:

```
LIB "hnoether.lib";
ring exring = 7,(x,y),ds;
list hne=develop(4x98+2x49y7+x11y14+2y14);
↦ a(0,7) = 3
↦ h(0) = 7
↦ h(1) = 1
↦ h(2) = 3
```

```
↦ h(3) = 1
↦ a(4,2) = -1
print(hne[1]);
↦ 0,0, 0,0,0,0,3,x,
↦ 0,x, 0,0,0,0,0,0,
↦ 0,0, 0,x,0,0,0,0,
↦ 0,x, 0,0,0,0,0,0,
↦ 0,-1,0,0,0,0,0,0
// therefore the HNE is:
// z(-1)= 3*z(0)^7 + z(0)^7*z(1),
// z(0) = z(1)*z(2),       (there is 1 zero in the 2nd row before x)
// z(1) = z(2)^3*z(3),     (there are 3 zeroes in the 3rd row)
// z(2) = z(3)*z(4),
// z(3) = -z(4)^2 + 0*z(4)^3 +...+ 0*z(4)^8 + ?*z(4)^9 + ...
// (the missing x in the last line indicates that it is not complete.)
hne[2];
↦ 7,1,3,1,-1
param(hne);
↦ // ** Warning: result is exact up to order 20 in x and 104 in y !
↦ _[1]=-x14
↦ _[2]=-3x98-x109
// parametrization:   x(t)= -t^14+O(t^21),  y(t)= -3t^98+O(t^105)
// (the term -t^109 in y may have a wrong coefficient)
displayHNE(hne);
↦ HNE[1]=-y+3*z(0)^7+z(0)^7*z(1)
↦ HNE[2]=-x+z(1)*z(2)
↦ HNE[3]=z(2)^3*z(3)
↦ HNE[4]=z(3)*z(4)
↦ HNE[5]=-z(4)^2
```

See also: Section D.5.5.7 [displayHNE], page 463; Section D.5.5.5 [extdevelop], page 460; Section D.5.5.3 [reddevelop], page 457.

D.5.5.3 reddevelop

Procedure from library `hnoether.lib` (see Section D.5.5 [hnoether_lib], page 454).

Usage: reddevelop(f); f poly

Assume: f is a bivariate polynomial (in the first 2 ring variables)

Create: ring with name `HNEring`, variables `x,y` and ordering `ls` over a field extension of the current basering's ground field.
Since the Hamburger-Noether development of a reducible curve singularity usually does not exist in the originally given basering, `reddevelop` always defines `HNEring` and CHANGES to it. The field extension is chosen minimally.

Return: list `L` of lists `L[i]` (corresponding to the output of `develop(f[i])`, f[i] a branch of f, but the last entry being omitted).

`L[i][1]`; matrix:
: Each row contains the coefficients of the corresponding line of the Hamburger-Noether expansion (HNE) for f[i]. The end of the line is marked in the matrix by the first ring variable (usually x).

`L[i][2]`; intvec:
: indicating the length of lines of the HNE

`L[i][3]`; int:
: 0 if the 1st ring variable was transversal (with respect to f[i]),
  1 if the variables were changed at the beginning of the computation,
  -1 if an error has occurred.

`L[i][4]`; poly:
: the transformed polynomial of f[i] to make it possible to extend the Hamburger-Noether development a posteriori without having to do all the previous calculation once again (0 if not needed)

Note: If `printlevel>=0` comments are displayed (default is `printlevel=0`).

Example:

```
LIB "hnoether.lib";
// -------- prepare for example ---------
if (nameof(basering)=="HNEring")
{
def rettering=HNEring;
kill HNEring;
}
// ------ the example starts here -------
ring r = 32003,(x,y),dp;
poly f = x25+x24-4x23-1x22y+4x22+8x21y-2x21-12x20y-4x19y2+4x20+10x19y
+12x18y2-24x18y-20x17y2-4x16y3+x18+60x16y2+20x15y3-9x16y
-80x14y3-10x13y4+36x14y2+60x12y4+2x11y5-84x12y3-24x10y5
+126x10y4+4x8y6-126x8y5+84x6y6-36x4y7+9x2y8-1y9;
list hne=reddevelop(f);
↦ finite HNE of one branch found
↦  Change of basering (field extension) necessary!
↦ HNE of one branch found
↦ HNE of one branch found
↦ HNE of one branch found
↦ // result: 4 branch(es) successfully computed,
↦ //         basering has changed to HNEring
size(hne);            // number of branches
↦ 4
print(hne[1][1]);     // HN-matrix of 1st branch
↦ 0,1,x,0,
↦ 0,x,0,0,
↦ 0,2,1,x,
↦ 0,1,x,0
print(hne[4][1]);     // HN-matrix of 4th branch
↦ 0,1,(-a+9380),(-4538a-14206)
// a ring change was necessary, a is a parameter
HNEring;
↦ //   characteristic : 32003
↦ //   1 parameter    : a
↦ //   minpoly        : (a2-9380a-6469)
↦ //   number of vars : 2
↦ //        block   1 : ordering ls
↦ //                  : names    x y
↦ //        block   2 : ordering C
kill HNEring,r;
// --- restore HNEring if previously defined ---
```

```
if (defined(rettering)) {
setring rettering;
def HNEring=rettering;
export HNEring;
}
```

See also: Section D.5.5.2 [develop], page 456; Section D.5.5.7 [displayHNE], page 463; Section D.5.5.4 [essdevelop], page 459; Section D.5.5.5 [extdevelop], page 460; Section D.5.5.6 [param], page 462.

D.5.5.4 essdevelop

Procedure from library `hnoether.lib` (see Section D.5.5 [hnoether_lib], page 454).

Usage: essdevelop(f); f poly

Assume: f is a bivariate polynomial (in the first 2 ring variables)

Create: ring with name `HNEring`, variables `x,y` and ordering `ls` over a field extension of the current basering's ground field.
Since the Hamburger-Noether development of a reducible curve singularity usually does not exist in the originally given basering, `essdevelop` always defines `HNEring` and CHANGES to it. The field extension is chosen minimally.

Return: list `L` of lists `L[i]` (corresponding to the output of `develop(f[i])`, f[i] an "essential" branch of f, but the last entry being omitted).
For more details type `help reddevelop;`.

Note: If the HNE needs a field extension, some of the branches will be conjugate. In this case `essdevelop` reduces the computation to one representative for each group of conjugate branches.
Note that the degree of each branch is in general less than the degree of the field extension in which all HNEs can be put.
Use `reddevelop` or `HNdevelop` to compute a complete HNE, i.e., a HNE for all branches.
If `printlevel>=0` comments are displayed (default is `printlevel=0`).

Example:

```
LIB "hnoether.lib";
// -------- prepare for example ---------
if (nameof(basering)=="HNEring") {
def rettering=HNEring;
kill HNEring;
}
// ------ the example starts here -------
ring r=2,(x,y),dp;
poly f=(x4+x2y+y2)*(x3+xy2+y3);
// --------- compute all branches: ---------
list hne=reddevelop(f);
↦  Change of basering (field extension) necessary!
↦ finite HNE found
↦ finite HNE found
↦ finite HNE found
↦  Change of basering (field extension) necessary!
↦ // new minimal polynomial: a6+a5+a3+a2+1
↦ finite HNE found
```

```
↦ finite HNE found
↦ // result: 5 branch(es) successfully computed,
↦ //         basering has changed to HNEring
displayHNE(hne[1]);   // HN-matrix of 1st branch
↦ HNE[1]=y+(a^4+a^2)*z(0)
↦ HNE[2]=x+z(0)
displayHNE(hne[4]);   // HN-matrix of 4th branch
↦ HNE[1]=y+(a^4+a^2+a)*z(0)^2
↦ HNE[2]=x+z(0)
setring r;
kill HNEring;
// --- compute only one of conjugate branches: ---
list hne=essdevelop(f);
↦  Change of basering (field extension) necessary!
↦ finite HNE found
↦  Change of basering (field extension) necessary!
↦ // new minimal polynomial: a6+a5+a3+a2+1
↦ finite HNE found
↦ // result: 2 branch(es) successfully computed;
↦ // note that conjugate branches are omitted and that the number
↦ // of branches represented by each remaining one may vary!
↦ // basering has changed to HNEring
displayHNE(hne);
↦ // Hamburger-Noether development of branch nr.1:
↦ HNE[1]=y+(a^4+a^2)*z(0)
↦ HNE[2]=x+z(0)
↦
↦ // Hamburger-Noether development of branch nr.2:
↦ HNE[1]=y+(a^4+a^2+a)*z(0)^2
↦ HNE[2]=x+z(0)
↦
// no. 1 of essdevelop represents no. 1 - 3 of reddevelop and
// no. 2 of essdevelop represents no. 4 + 5 of reddevelop
kill HNEring,r;
// --- restore HNEring if previously defined ---
if (defined(rettering)) {
setring rettering;
def HNEring=rettering;
export HNEring;
}
```

See also: Section D.5.5.1 [HNdevelop], page 454; Section D.5.5.2 [develop], page 456; Section D.5.5.5 [extdevelop], page 460; Section D.5.5.3 [reddevelop], page 457.

D.5.5.5 extdevelop

Procedure from library `hnoether.lib` (see Section D.5.5 [hnoether_lib], page 454).

Usage: extdevelop(L,N); list L, int N

Assume: L is the output of `develop(f)`, or of `extdevelop(l,n)`, or one entry of the output of `reddevelop(f)`.

Return: an extension of the Hamburger-Noether development of f as a list in the same format as L has (up to the last entry in the output of `develop(f)`).
Type `help develop;`, resp. `help reddevelop;` for more details.

Note: The new HN-matrix will have at least N columns (if the HNE is not finite). In particular, if f is irreducible then (in most cases) `extdevelop(develop(f),N)` will produce the same result as `develop(f,N)`.
If the matrix M of L has n columns then, compared with `param(L)`, `param(extdevelop(L,N))` will increase the exactness by at least (N-n) more significant monomials.

Example:

```
LIB "hnoether.lib";
// -------- prepare for example ---------
if (nameof(basering)=="HNEring") {
def rettering=HNEring;
kill HNEring;
}
// ------ the example starts here -------
ring exring=0,(x,y),dp;
list hne=reddevelop(x14-3y2x11-y3x10-y2x9+3y4x8+y5x7+3y4x6+x5*(-y6+y5)
-3y6x3-y7x2+y8);
↦ finite HNE found
↦ HNE of one branch found
↦ finite HNE found
↦ // result: 3 branch(es) successfully computed,
↦ //         basering has changed to HNEring
print(hne[1][1]);    // HNE of 1st branch is finite
↦ 0,x,0,
↦ 0,1,x
print(extdevelop(hne[1],5)[1]);
↦  No extension is possible
↦ 0,x,0,
↦ 0,1,x
print(hne[2][1]);    // HNE of 2nd branch can be extended
↦ 0,x,0,
↦ 0,1,x,
↦ 0,1,-1
list ehne=extdevelop(hne[2],5);
↦ a(2,3) = -1
↦ a(2,4) = 1
↦ a(2,5) = -1
print(ehne[1]);      // new HN-matrix has 5 columns
↦ 0,x,0, 0,0,
↦ 0,1,x, 0,0,
↦ 0,1,-1,1,-1
param(hne[2]);
↦ // ** Warning: result is exact up to order 5 in x and 7 in y !
↦ _[1]=x4-x5-x6+x7
↦ _[2]=x6-2x7+2x9-x10
param(ehne);
↦ // ** Warning: result is exact up to order 7 in x and 9 in y !
↦ _[1]=x4-x5+x6-x7-x8+x9-x10+x11
↦ _[2]=x6-2x7+3x8-4x9+2x10-2x12+4x13-3x14+2x15-x16
kill HNEring,exring;
// --- restore HNEring if previously defined ---
if (defined(rettering)) {
```

```
setring rettering;
def HNEring=rettering;
export HNEring;
}
```

See also: Section D.5.5.2 [develop], page 456; Section D.5.5.6 [param], page 462; Section D.5.5.3 [reddevelop], page 457.

D.5.5.6 param

Procedure from library `hnoether.lib` (see Section D.5.5 [hnoether_lib], page 454).

Usage: param(L [,x]); L list, x any type (optional)

Assume: L is the output of `develop(f)`, or of `extdevelop(develop(f),n)`, or one entry of the output of `reddevelop(f)`.

Return: a parametrization for f in the following format:
- if only the list L is given, the result is an ideal of two polynomials p[1],p[2]: if the HNE was finite then f(p[1],p[2])=0}; if not, the "real" parametrization will be two power series and p[1],p[2] are truncations of these series.
- if the optional parameter x is given, the result is a list l: l[1]=param(L) (ideal) and l[2]=intvec with two entries indicating the highest degree up to which the coefficients of the monomials in l[1] are exact (entry -1 means that the corresponding parametrization is exact).

Note: If the basering has only 2 variables, the first variable is chosen as indefinite. Otherwise, the 3rd variable is chosen.

Example:

```
LIB "hnoether.lib";
ring exring=0,(x,y,t),ds;
poly f=x3+2xy2+y2;
list hne=develop(f);
↦  Warning! You have defined too many variables!
↦  All variables except the first two will be ignored!
↦ h(0) = 1
↦ a(1,2) = -1
list hne_extended=extdevelop(hne,10);
↦ a(1,4) = 2
↦ a(1,6) = -4
↦ a(1,8) = 8
↦ a(1,10) = -16
//   compare the matrices ...
print(hne[1]);
↦ 0,x,
↦ 0,-1
print(hne_extended[1]);
↦ 0,x, 0,0,0,0, 0,0,0,0,
↦ 0,-1,0,2,0,-4,0,8,0,-16
// ... and the resulting parametrizations:
param(hne);
↦ // ** Warning: result is exact up to order 2 in x and 3 in y !
↦ _[1]=-t2
↦ _[2]=-t3
```

```
param(hne_extended);
↦ // ** Warning: result is exact up to order 10 in x and 11 in y !
↦ _[1]=-t2+2t4-4t6+8t8-16t10
↦ _[2]=-t3+2t5-4t7+8t9-16t11
param(hne_extended,0);
↦ // ** Warning: result is exact up to order 10 in x and 11 in y !
↦ [1]:
↦    _[1]=-t2+2t4-4t6+8t8-16t10
↦    _[2]=-t3+2t5-4t7+8t9-16t11
↦ [2]:
↦    10,11
```

See also: Section D.5.5.2 [develop], page 456; Section D.5.5.5 [extdevelop], page 460.

D.5.5.7 displayHNE

Procedure from library `hnoether.lib` (see Section D.5.5 [hnoether_lib], page 454).

Usage: displayHNE(L[,n]); L list, n int

Assume: L has the format of the output of `develop(f)`, resp. of `reddevelop(f)`.

Return: - if only one argument is given, no return value, but display an ideal HNE of the following form:

```
HNE[1]=-y+[]*z(0)^1+[]*z(0)^2+...+z(0)^<>*z(1)
HNE[2]=-x+          []*z(1)^2+...+z(1)^<>*z(2)
HNE[3]=             []*z(2)^2+...+z(2)^<>*z(3)
.......             .........................
HNE[r+1]=           []*z(r)^2+[]*z(r)^3+......
```

where `x`,`y` are the first 2 variables of the basering. The values of `[]` are the coefficients of the Hamburger-Noether matrix, the values of `<>` are represented by `x` in the HN-matrix.
- if a second argument is given, create and export a new ring with name `displayring` containing an ideal `HNE` as described above.
- if L corresponds to the output of `reddevelop(f)`, `displayHNE(L[,n])` shows the HNE's of all branches of f in the form described above. The optional parameter is then ignored.

Note: The 1st line of the above ideal (i.e., `HNE[1]`) means that `y=[]*z(0)^1+...`, the 2nd line (`HNE[2]`) means that `x=[]*z(1)^2+...`, so you can see which indeterminate corresponds to which line (it's also possible that `x` corresponds to the 1st line and `y` to the 2nd).

Example:

```
LIB "hnoether.lib";
ring r=0,(x,y),dp;
poly f=x3+2xy2+y2;
list hn=develop(f);
↦ h(0) = 1
↦ a(1,2) = -1
displayHNE(hn);
↦ HNE[1]=-y+z(0)*z(1)
↦ HNE[2]=-x-z(1)^2
```

See also: Section D.5.5.2 [develop], page 456; Section D.5.5.3 [reddevelop], page 457.

D.5.5.8 invariants

Procedure from library `hnoether.lib` (see Section D.5.5 [hnoether_lib], page 454).

Usage: invariants(L); L list

Assume: L is the output of `develop(f)`, or of `extdevelop(develop(f),n)`, or one entry of the output of `reddevelop(f)`.

Return: list, if L contains a valid HNE:

invariants(L)[1]:	intvec	(characteristic exponents)
invariants(L)[2]:	intvec	(generators of the semigroup)
invariants(L)[3]:	intvec	(Puiseux pairs, 1st components)
invariants(L)[4]:	intvec	(Puiseux pairs, 2nd components)
invariants(L)[5]:	int	(degree of the conductor)
invariants(L)[6]:	intvec	(sequence of multiplicities)

an empty list, if L contains no valid HNE.

Example:

```
LIB "hnoether.lib";
ring exring=0,(x,y),dp;
list hne=develop(y4+2x3y2+x6+x5y);
↦ h(0) = 1
↦ a(1,2) = -1
↦ h(1) = 2
↦ a(2,2) = -1
↦ a(2,3) = -2
list erg=invariants(hne);
erg[1];                   // the characteristic exponents
↦ 4,6,7
erg[2];                   // the generators of the semigroup of values
↦ 4,6,13
erg[3],erg[4];            // the Puiseux pairs in packed form
↦ 3,7 2,2
erg[5] / 2;               // the delta-invariant
↦ 8
erg[6];                   // the sequence of multiplicities
↦ 4,2,2,1
// To display the invariants more 'nicely':
displayInvariants(hne);
↦  characteristic exponents  : 4,6,7
↦  generators of semigroup   : 4,6,13
↦  Puiseux pairs             : (3,2)(7,2)
↦  degree of the conductor   : 16
↦  delta invariant           : 8
↦  sequence of multiplicities: 4,2,2,1
```

See also: Section D.5.5.2 [develop], page 456; Section D.5.5.10 [displayInvariants], page 466; Section D.5.5.11 [intersection], page 466; Section D.5.5.9 [multsequence], page 464.

D.5.5.9 multsequence

Procedure from library `hnoether.lib` (see Section D.5.5 [hnoether_lib], page 454).

Usage: multsequence(L); L list

Assume: L is the output of `develop(f)`, or of `extdevelop(develop(f),n)`, or one entry of the output of `reddevelop(f)`.

Return: intvec corresponding to the multiplicity sequence of (a branch) of the curve (the same as `invariants(L)[6]`).

Assume: L is the output of `reddevelop(f)`.

Return: list of two integer matrices:

`multsequence(L)[1][i,*]`
: contains the multiplicities of the branches at their infinitely near point of 0 in its (i-1) order neighbourhood (i.e., i=1: multiplicity of the branches themselves, i=2: multiplicity of their 1st quadratic transformed, etc., Hence, `multsequence(L)[1][*,j]` is the multiplicity sequence of branch j.

`multsequence(L)[2][i,*]`:
: contains the information which of these infinitely near points coincide.

Note: The order of elements of the list obtained from `reddevelop(f)` must not be changed (because then the coincident infinitely near points couldn't be grouped together, cf. meaning of 2nd intmat in example). Hence, it is not wise to compute the HNE of several polynomials separately, put them into a list L and call `multsequence(L)`.
Use `displayMultsequence` to produce a better readable output for reducible curves on the screen.

Example:

```
LIB "hnoether.lib";
// -------- prepare for example ---------
if (nameof(basering)=="HNEring") {
def rettering=HNEring;
kill HNEring;
}
// ------ the example starts here -------
int plevel=printlevel; printlevel=-1;
ring r=0,(x,y),dp;
list hne=reddevelop((x6-y10)*(x+y2-y3)*(x+y2+y3));   // 4 branches
multsequence(hne[1]),"  |  ",multsequence(hne[2]),"  |  ",
multsequence(hne[3]),"  |  ",multsequence(hne[4]);
↦ 3,2,1   |   3,2,1   |   1   |   1
multsequence(hne);
↦ [1]:
↦    3,3,1,1,
↦    2,2,1,1,
↦    1,1,1,1,
↦    1,1,1,1,
↦    1,1,1,1
↦ [2]:
↦    4,0,0,0,
↦    4,0,0,0,
↦    2,2,0,0,
↦    2,1,1,0,
↦    1,1,1,1
// The meaning of the entries of the 2nd matrix is as follows:
displayMultsequence(hne);
```

```
↦ [(3,3,1,1)],
↦ [(2,2,1,1)],
↦ [(1,1),(1,1)],
↦ [(1,1),(1),(1)],
↦ [(1),(1),(1),(1)]
kill HNEring,r;
printlevel=plevel;
// --- restore HNEring if previously defined ---
if (defined(rettering)) {
setring rettering;
def HNEring=rettering;
export HNEring;
}
```

See also: Section D.5.5.2 [develop], page 456; Section D.5.5.12 [displayMultsequence], page 467; Section D.5.5.3 [reddevelop], page 457; Section D.5.5.19 [separateHNE], page 471.

D.5.5.10 displayInvariants

Procedure from library `hnoether.lib` (see Section D.5.5 [hnoether_lib], page 454).

Usage: displayInvariants(L); L list

Assume: L is the output of `develop(f)`, or of `extdevelop(develop(f),n)`, or (one entry of) the output of `reddevelop(f)`.

Return: none

Display: invariants of the corresponding branch, resp. of all branches, in a better readable form.

Example:

```
LIB "hnoether.lib";
ring exring=0,(x,y),dp;
list hne=develop(y4+2x3y2+x6+x5y);
↦ h(0) = 1
↦ a(1,2) = -1
↦ h(1) = 2
↦ a(2,2) = -1
↦ a(2,3) = -2
displayInvariants(hne);
↦  characteristic exponents  : 4,6,7
↦  generators of semigroup   : 4,6,13
↦  Puiseux pairs             : (3,2)(7,2)
↦  degree of the conductor   : 16
↦  delta invariant           : 8
↦  sequence of multiplicities: 4,2,2,1
```

See also: Section D.5.5.2 [develop], page 456; Section D.5.5.11 [intersection], page 466; Section D.5.5.8 [invariants], page 464; Section D.5.5.3 [reddevelop], page 457.

D.5.5.11 intersection

Procedure from library `hnoether.lib` (see Section D.5.5 [hnoether_lib], page 454).

Usage: intersection(hne1,hne2); hne1, hne2 lists

Assume: hne1, hne2 represent a HNE (i.e., are the output of `develop(f)`, or of `extdevelop(develop(f),n)`, or one entry of the output of `reddevelop(f)`).

Return: int, the intersection multiplicity of the branches corresponding to hne1 and hne2.

Example:

```
LIB "hnoether.lib";
// -------- prepare for example ---------
if (nameof(basering)=="HNEring") {
def rettering=HNEring;
kill HNEring;
}
// ------ the example starts here -------
int plevel=printlevel; printlevel=-1;
ring r=0,(x,y),dp;
list hne=reddevelop((x2-y3)*(x2+y3));
intersection(hne[1],hne[2]);
↦ 6
kill HNEring,r;
printlevel=plevel;
// --- restore HNEring if previously defined ---
if (defined(rettering)) {
setring rettering;
def HNEring=rettering;
export HNEring;
}
```

See also: Section D.5.5.10 [displayInvariants], page 466; Section D.5.5.3 [reddevelop], page 457.

D.5.5.12 displayMultsequence

Procedure from library `hnoether.lib` (see Section D.5.5 [hnoether_lib], page 454).

Usage: displayMultsequence(L);

Assume: L is the output of `develop(f)`, or of `extdevelop(develop(f),n)`, or (one entry of) the output of `reddevelop(f)`.

Return: nothing

Display: the sequence of multiplicities:

- if `L=develop(f)` or `L=reddevelop(f)[i]`:

```
a , b , c , ....... , 1
```

- if `L=reddevelop(f)`:

```
[(a_1, .... , b_1 , .... , c_1)],
[(a_2, ... ), ... , (... , c_2)],
 ...................................... ,
[(a_n),(b_n), ....., (c_n)]
```

with:

`a_1 , ... , a_n` the sequence of multiplicities of the 1st branch,

`[...]` the multiplicities of the j-th transformed of all branches,

`(...)` indicating branches meeting in an infinitely near point.

Note: The same restrictions for L as in `multsequence` apply.

Example:

```
LIB "hnoether.lib";
// -------- prepare for example ---------
if (nameof(basering)=="HNEring") {
```

```
def rettering=HNEring;
kill HNEring;
}
// ------ the example starts here -------
int plevel=printlevel; printlevel=-1;
ring r=0,(x,y),dp;
list hne=reddevelop((x6-y10)*(x+y2-y3)*(x+y2+y3));
displayMultsequence(hne[1]);
↦ The sequence of multiplicities is   3,2,1
displayMultsequence(hne);
↦ [(3,3,1,1)],
↦ [(2,2,1,1)],
↦ [(1,1),(1,1)],
↦ [(1,1),(1),(1)],
↦ [(1),(1),(1),(1)]
kill HNEring,r;
printlevel=plevel;
// --- restore HNEring if previously defined ---
if (defined(rettering)) {
setring rettering;
def HNEring=rettering;
export HNEring;
}
```

See also: Section D.5.5.2 [develop], page 456; Section D.5.5.9 [multsequence], page 464; Section D.5.5.3 [reddevelop], page 457; Section D.5.5.19 [separateHNE], page 471.

D.5.5.13 stripHNE

Procedure from library `hnoether.lib` (see Section D.5.5 [hnoether_lib], page 454).

Usage: stripHNE(L); L list

Assume: L is the output of `develop(f)`, or of `extdevelop(develop(f),n)`, or (one entry of) the output of `reddevelop(f)`.

Return: list in the same format as L, but all polynomials L[4], resp. L[i][4], are set to zero.

Note: The purpose of this procedure is to remove huge amounts of data no longer needed. It is useful, if one or more of the polynomials in L consume much memory. It is still possible to compute invariants, parametrizations etc. with the stripped HNE(s), but it is not possible to use `extdevelop` with them.

Example:

```
LIB "hnoether.lib";
int plevel=printlevel; printlevel=-1;
ring r=0,(x,y),dp;
list hne=develop(x2+y3+y4);
hne;
↦ [1]:
↦    _[1,1]=0
↦    _[1,2]=x
↦    _[2,1]=0
↦    _[2,2]=-1
↦ [2]:
↦    1,-1
```

```
↦ [3]:
↦    1
↦ [4]:
↦    x4-2x2y+y2+y
↦ [5]:
↦    1
stripHNE(hne);
↦ [1]:
↦    _[1,1]=0
↦    _[1,2]=x
↦    _[2,1]=0
↦    _[2,2]=-1
↦ [2]:
↦    1,-1
↦ [3]:
↦    1
↦ [4]:
↦    0
↦ [5]:
↦    1
printlevel=plevel;
```

See also: Section D.5.5.2 [develop], page 456; Section D.5.5.5 [extdevelop], page 460; Section D.5.5.3 [reddevelop], page 457.

D.5.5.14 is_irred

Procedure from library `hnoether.lib` (see Section D.5.5 [hnoether_lib], page 454).

Usage: is_irred(f); f poly

Assume: f is a squarefree bivariate polynomial (in the first 2 ring variables).

Return: int (0 or 1):
- `is_irred(f)=1` if f is irreducible as a formal power series over the algebraic closure of its coefficient field (f defines an analytically irreducible curve at zero),
- `is_irred(f)=0` otherwise.

Note: 0 and units in the ring of formal power series are considered to be not irreducible.

Example:

```
LIB "hnoether.lib";
ring exring=0,(x,y),ls;
is_irred(x2+y3);
↦ 1
is_irred(x2+y2);
↦ 0
is_irred(x2+y3+1);
↦ 0
```

D.5.5.15 deltaHNE

Procedure from library `hnoether.lib` (see Section D.5.5 [hnoether_lib], page 454).

D.5.5.16 delta

Procedure from library `hnoether.lib` (see Section D.5.5 [hnoether_lib], page 454).

Usage: delta(f); f a polynomial defining an isolated plane curve singularity at 0

Return: the delta invariant of the singularity at 0, the vector space dimension of R~/R, where R~ is the normalization of the singularity R=basering/f

Example:

```
LIB "hnoether.lib";
ring r = 32003,(x,y),ds;
poly f = x25+x24-4x23-1x22y+4x22+8x21y-2x21-12x20y-4x19y2+4x20+10x19y
+12x18y2-24x18y-20x17y2-4x16y3+x18+60x16y2+20x15y3-9x16y
-80x14y3-10x13y4+36x14y2+60x12y4+2x11y5-84x12y3-24x10y5
+126x10y4+4x8y6-126x8y5+84x6y6-36x4y7+9x2y8-1y9;
delta(f);
↦ finite HNE of one branch found
↦  Change of basering (field extension) necessary!
↦ HNE of one branch found
↦ HNE of one branch found
↦ HNE of one branch found
↦ // result: 4 branch(es) successfully computed,
↦ //         basering has changed to HNEring
↦ 96
```

D.5.5.17 puiseux2generators

Procedure from library `hnoether.lib` (see Section D.5.5 [hnoether_lib], page 454).

Usage: puiseux2generators(m,n); m,n intvec

Assume: m, resp. n, represent the 1st, resp. 2nd, components of Puiseux pairs (e.g., `m=invariants(L)[3]`, `n=invariants(L)[4]`).

Return: intvec of the generators of the semigroup of values.

Example:

```
LIB "hnoether.lib";
// take (3,2),(7,2),(15,2),(31,2),(63,2),(127,2) as Puiseux pairs:
puiseux2generators(intvec(3,7,15,31,63,127),intvec(2,2,2,2,2,2));
↦ 64,96,208,424,852,1706,3413
```

See also: Section D.5.5.8 [invariants], page 464.

D.5.5.18 multiplicities

Procedure from library `hnoether.lib` (see Section D.5.5 [hnoether_lib], page 454).

Usage: multiplicities(L); L list

Assume: L is the output of `develop(f)`, or of `extdevelop(develop(f),n)`, or one entry of the output of `reddevelop(f)`.

Return: intvec of the different multiplicities that occur when successively blowing-up the curve singularity corresponding to f.

Example:

```
LIB "hnoether.lib";
int p=printlevel; printlevel=-1;
ring r=0,(x,y),dp;
multiplicities(develop(x5+y7));
↦ 5,2,1
// The first value is the multiplicity of the curve itself, here it's 5
printlevel=p;
```

See also: Section D.5.5.2 [develop], page 456.

D.5.5.19 separateHNE

Procedure from library `hnoether.lib` (see Section D.5.5 [hnoether_lib], page 454).

Usage: separateHNE(hne1,hne2); hne1, hne2 lists

Assume: hne1, hne2 are HNEs (=output of `develop(f)`, `extdevelop(develop(f),n)`, or one entry of the output of `reddevelop(f)`).

Return: number of quadratic transformations needed to separate both curves (branches).

Example:

```
LIB "hnoether.lib";
int p=printlevel; printlevel=-1;
ring r=0,(x,y),dp;
list hne1=develop(x);
list hne2=develop(x+y);
list hne3=develop(x+y2);
separateHNE(hne1,hne2);  // two transversal lines
↦ 1
separateHNE(hne1,hne3);  // one quadratic transform. gives 1st example
↦ 2
printlevel=p;
```

See also: Section D.5.5.2 [develop], page 456; Section D.5.5.12 [displayMultsequence], page 467; Section D.5.5.9 [multsequence], page 464; Section D.5.5.3 [reddevelop], page 457.

D.5.5.20 squarefree

Procedure from library `hnoether.lib` (see Section D.5.5 [hnoether_lib], page 454).

Usage: squarefree(f); f poly

Assume: f is a bivariate polynomial (in the first 2 ring variables).

Return: poly, a squarefree divisor of f.

Note: Usually, the return value is the greatest squarefree divisor, but there is one exception: factors with a p-th root, p the characteristic of the basering, are lost.

Example:

```
LIB "hnoether.lib";
ring exring=3,(x,y),dp;
squarefree((x3+y)^2);
↦ x3+y
squarefree((x+y)^3*(x-y)^2); // Warning: (x+y)^3 is lost
↦ x-y
squarefree((x+y)^4*(x-y)^2); // result is (x+y)*(x-y)
↦ x2-y2
```

See also: Section D.5.5.21 [allsquarefree], page 472.

D.5.5.21 allsquarefree

Procedure from library `hnoether.lib` (see Section D.5.5 [hnoether_lib], page 454).

Usage : allsquarefree(f,g); f,g poly

Assume: g is the output of `squarefree(f)`.

Return: the greatest squarefree divisor of f.

Note : This proc uses factorize to get the missing factors of f not in g and, therefore, may be slow.

Example:

```
LIB "hnoether.lib";
ring exring=7,(x,y),dp;
poly f=(x+y)^7*(x-y)^8;
poly g=squarefree(f);
g;                      // factor x+y lost, since characteristic=7
↦ x-y
allsquarefree(f,g);     // all factors (x+y)*(x-y) found
↦ x2-y2
```

See also: Section D.5.5.20 [squarefree], page 471.

D.5.5.22 further_hn_proc

Procedure from library `hnoether.lib` (see Section D.5.5 [hnoether_lib], page 454).

Usage: further_hn_proc();

Note: The library `hnoether.lib` contains some more procedures which are not shown when typing `help hnoether.lib;`. They may be useful for interactive use (e.g. if you want to do the calculation of an HN development "by hand" to see the intermediate results), and they can be enumerated by calling `further_hn_proc()`.
Use `help <procedure>;` for detailed information about each of them.

Example:

```
LIB "hnoether.lib";
further_hn_proc();
↦
↦  The following procedures are also part of `hnoether.lib':
↦
↦  newtonpoly(f);      Newton polygon of polynom f
↦  getnm(f);           intersection pts. of Newton polygon with axes
↦  T_Transform(f,Q,N); returns f(y,xy^Q)/y^NQ (f: poly, Q,N: int)
↦  T1_Transform(f,d,M); returns f(x,y+d*x^M)  (f: poly,d:number,M:int)
↦  T2_Transform(f,d,M,N,ref);   a composition of T1 & T
↦  koeff(f,I,J);       gets coefficient of indicated monomial of poly f
↦  redleit(f,S,E);     restriction of monomials of f to line (S-E)
↦  leit(f,n,m);        special case of redleit (for irred. polynomials)
↦  testreducible(f,n,m); tests whether f is reducible
↦  charPoly(f,M,N);    characteristic polynomial of f
↦  find_in_list(L,p);  find int p in list L
↦  get_last_divisor(M,N); last divisor in Euclid's algorithm
↦  factorfirst(f,M,N); try to factor f without `factorize'
↦  factorlist(L);      factorize a list L of polynomials
↦  referencepoly(D);   a polynomial f s.t. D is the Newton diagram of f
```

D.5.6 mondromy_lib

Library: mondromy.lib

Purpose: Monodromy of an Isolated Hypersurface Singularity

Author: Mathias Schulze, email: mschulze@mathematik.uni-kl.de

Overview: A library to compute the monodromy of an isolated hypersurface singularity. It uses an algorithm by Brieskorn (manuscripta math. 2 (1970), 103-161) to compute a connection matrix of the meromorphic Gauss-Manin connection up to arbitrarily high order, and an algorithm of Gerard and Levelt (Ann. Inst. Fourier, Grenoble 23,1 (1973), pp. 157-195) to transform it to a simple pole.

Procedures: **See also:** Section D.5.4 [gaussman_lib], page 440.

D.5.6.1 detadj

Procedure from library `mondromy.lib` (see Section D.5.6 [mondromy_lib], page 473).

Usage: detadj(U); U matrix

Assume: U is a square matrix with non zero determinant.

Return: The procedure returns a list with at most 2 entries.
If U is not a square matrix, the list is empty.
If U is a square matrix, then the first entry is the determinant of U. If U is a square matrix and the determinant of U not zero, then the second entry is the adjoint matrix of U.

Display: The procedure displays comments if printlevel>=1.

Example:

```
LIB "mondromy.lib";
ring R=0,x,dp;
matrix U[2][2]=1,1+x,1+x2,1+x3;
list daU=detadj(U);
daU[1];
↦ -x2-x
print(daU[2]);
↦ x3+1, -x-1,
↦ -x2-1,1
```

D.5.6.2 invunit

Procedure from library `mondromy.lib` (see Section D.5.6 [mondromy_lib], page 473).

Usage: invunit(u,n); u poly, n int

Assume: The polynomial u is a series unit.

Return: The procedure returns the series inverse of u up to order n or a zero polynomial if u is no series unit.

Display: The procedure displays comments if printlevel>=1.

Example:

```
LIB "mondromy.lib";
ring R=0,(x,y),dp;
invunit(2+x3+xy4,10);
↦ 1/8x2y8-1/16x9+1/4x4y4+1/8x6-1/4xy4-1/4x3+1/2
```

D.5.6.3 jacoblift

Procedure from library `mondromy.lib` (see Section D.5.6 [mondromy_lib], page 473).

Usage: jacoblift(f); f poly

Assume: The polynomial f in a series ring (local ordering) defines an isolated hypersurface singularity.

Return: The procedure returns a list with entries kappa, xi, u of type int, vector, poly such that kappa is minimal with f^kappa in jacob(f), u is a unit, and u*f^kappa=(matrix(jacob(f))*xi)[1,1].

Display: The procedure displays comments if printlevel>=1.

Example:

```
LIB "mondromy.lib";
ring R=0,(x,y),ds;
poly f=x2y2+x6+y6;
jacoblift(f);
↦ [1]:
↦    2
↦ [2]:
↦    1/2x2y3*gen(2)+1/6x7*gen(1)+5/6x6y*gen(2)-2/3xy6*gen(1)+1/6y7*gen(2)-4\
   x4y5*gen(2)-3/2x9y2*gen(1)-15/2x8y3*gen(2)+9/2x3y8*gen(1)-3/2x2y9*gen(2)
↦ [3]:
↦    1-9x2y2
```

D.5.6.4 monodromyB

Procedure from library `mondromy.lib` (see Section D.5.6 [mondromy_lib], page 473).

Usage: monodromyB(f[,opt]); f poly, opt int

Assume: The polynomial f in a series ring (local ordering) defines an isolated hypersurface singularity.

Return: The procedure returns a residue matrix M of the meromorphic Gauss-Manin connection of the singularity defined by f or an empty matrix if the assumptions are not fulfilled. If opt=0 (default), exp(-2*pi*i*M) is a monodromy matrix of f, else, only the characteristic polynomial of exp(-2*pi*i*M) coincides with the characteristic polynomial of the monodromy of f.

Display: The procedure displays more comments for higher printlevel.

Example:

```
LIB "mondromy.lib";
ring R=0,(x,y),ds;
poly f=x2y2+x6+y6;
matrix M=monodromyB(f);
print(M);
↦ 7/6,0,  0,0,  0,  0,0,   0,-1/2,0,  0,  0,    0,
↦ 0,  7/6,0,0,  0,  0,-1/2,0,0,   0,  0,  0,    0,
↦ 0,  0,  1,0,  0,  0,0,   0,0,   0,  0,  0,    0,
↦ 0,  0,  0,4/3,0,  0,0,   0,0,   0,  0,  0,    0,
↦ 0,  0,  0,0,  4/3,0,0,   0,0,   0,  0,  0,    0,
↦ 0,  0,  0,0,  0,  1,0,   0,0,   0,  0,  0,    0,
```

```
↦ 0,  0,  0,0,  0,  0,5/6, 0,0,   0,  0,  0,    0,
↦ 0,  0,  0,0,  0,  0,0,   1,0,   0,  0,  0,    0,
↦ 0,  0,  0,0,  0,  0,0,   0,5/6, 0,  0,  0,    0,
↦ 0,  0,  0,0,  0,  0,0,   0,0,   2/3,0,  0,    0,
↦ 0,  0,  0,0,  0,  0,0,   0,0,   0,  2/3,0,    0,
↦ 0,  0,  0,0,  0,  0,0,   0,0,   0,  0,  47/44,-625/396,
↦ 0,  0,  0,0,  0,  0,0,   0,0,   0,  0,  9/44, -3/44
```

D.5.6.5 H2basis

Procedure from library `mondromy.lib` (see Section D.5.6 [mondromy_lib], page 473).

Usage: H2basis(f); f poly

Assume: The polynomial f in a series ring (local ordering) defines an isolated hypersurface singularity.

Return: The procedure returns a list of representatives of a C{f}-basis of the Brieskorn lattice H"=Omega^(n+1)/df^dOmega^(n-1).

Theory: H" is a free C{f}-module of rank milnor(f).

Display: The procedure displays more comments for higher printlevel.

Example:

```
LIB "mondromy.lib";
ring R=0,(x,y),ds;
poly f=x2y2+x6+y6;
H2basis(f);
↦ [1]:
↦    x4
↦ [2]:
↦    x2y2
↦ [3]:
↦    y4
↦ [4]:
↦    x3
↦ [5]:
↦    x2y
↦ [6]:
↦    xy2
↦ [7]:
↦    y3
↦ [8]:
↦    x2
↦ [9]:
↦    xy
↦ [10]:
↦    y2
↦ [11]:
↦    x
↦ [12]:
↦    y
↦ [13]:
↦    1
```

D.5.7 qhmoduli_lib

Library: qhmoduli.lib

Purpose: Moduli Spaces of Semi-Quasihomogeneous Singularities

Author: Thomas Bayer, email: bayert@in.tum.de

Procedures:

D.5.7.1 ArnoldAction

Procedure from library `qhmoduli.lib` (see Section D.5.7 [qhmoduli_lib], page 476).

Usage: ArnoldAction(f, [Gf, B]); poly f; list Gf, B;
'Gf' is a list of two rings (coming from 'StabEqn')

Purpose: compute the induced action of the stabilizer G of f on T_, where T_ is given by the upper monomials B of the Milnor algebra of f.

Assume: f is quasihomogeneous

Return: polynomial ring over the same ground field, containing the ideals 'actionid' and 'stabid'.
- 'actionid' is the ideal defining the induced action of Gf on T_
- 'stabid' is the ideal of the stabilizer Gf in the new ring

Example:

```
LIB "qhmoduli.lib";
ring B   = 0,(x,y,z), ls;
poly f = -z5+y5+x2z+x2y;
def R = ArnoldAction(f);
setring R;
actionid;
↦ actionid[1]=-s(2)*t(1)+s(3)*t(1)
↦ actionid[2]=-s(2)^2*t(2)+2*s(2)^2*t(3)^2+s(3)^2*t(2)
↦ actionid[3]=s(2)*t(3)+s(3)*t(3)
stabid;
↦ stabid[1]=s(2)*s(3)
↦ stabid[2]=s(1)^2*s(2)+s(1)^2*s(3)-1
↦ stabid[3]=s(1)^2*s(3)^2-s(3)
↦ stabid[4]=s(1)^2+s(2)^4-s(3)^4
↦ stabid[5]=s(1)^4+s(2)^3-s(3)^3
↦ stabid[6]=-s(1)^2*s(3)+s(3)^5
```

D.5.7.2 ModEqn

Procedure from library `qhmoduli.lib` (see Section D.5.7 [qhmoduli_lib], page 476).

Usage: ModEqn(f [, opt]); poly f; int opt;

Purpose: compute equations of the moduli space of semiquasihomogeneous hypersurface singularity with principal part f w.r.t. right equivalence

Assume: f quasihomogeneous polynomial with an isolated singularity at 0

Return: polynomial ring, possibly a simple extension of the ground field of the basering, containing the ideal 'modid'

- 'modid' is the ideal of the moduli space if opt is even (> 0). otherwise it contains generators of the coordinate ring R of the moduli space (note : Spec(R) is the moduli space)

Options: 1 compute equations of the mod. space,
2 use a primary decomposition
4 compute E_f0, i.e., the image of G_f0
To combine options, add their value, default: opt =7

Example:

```
LIB "qhmoduli.lib";
ring B   = 0,(x,y), ls;
poly f = -x4 + xy5;
def R = ModEqn(f);
setring R;
modid;
↦ modid[1]=Y(5)^2-Y(4)*Y(6)
↦ modid[2]=Y(4)*Y(5)-Y(3)*Y(6)
↦ modid[3]=Y(3)*Y(5)-Y(2)*Y(6)
↦ modid[4]=Y(2)*Y(5)-Y(1)*Y(6)
↦ modid[5]=Y(4)^2-Y(2)*Y(6)
↦ modid[6]=Y(3)*Y(4)-Y(1)*Y(6)
↦ modid[7]=Y(2)*Y(4)-Y(1)*Y(5)
↦ modid[8]=Y(3)^2-Y(1)*Y(5)
↦ modid[9]=Y(2)*Y(3)-Y(1)*Y(4)
↦ modid[10]=Y(2)^2-Y(1)*Y(3)
```

D.5.7.3 QuotientEquations

Procedure from library `qhmoduli.lib` (see Section D.5.7 [qhmoduli_lib], page 476).

Usage: QuotientEquations(G,action,emb [, opt]); ideal G,action,emb;int opt

Purpose: compute the quotient of the variety given by the parameterization 'emb' by the linear action 'action' of the algebraic group G.

Assume: 'action' is linear, G must be finite if the Reynolds operator is needed (i.e., NullCone(G,action) returns some non-invariant polys)

Return: polynomial ring over a simple extension of the ground field of the basering, containing the ideals 'id' and 'embedid'.
- 'id' contains the equations of the quotient, if opt = 1; if opt = 0, 2, 'id' contains generators of the coordinate ring R of the quotient (Spec(R) is the quotient)
- 'embedid' = 0, if opt = 1;
if opt = 0, 2, it is the ideal defining the equivariant embedding

Options: 1 compute equations of the quotient,
2 use a primary decomposition when computing the Reynolds operator
To combine options, add their value, default: opt =3.

D.5.7.4 StabEqn

Procedure from library `qhmoduli.lib` (see Section D.5.7 [qhmoduli_lib], page 476).

Usage: StabEqn(f); f polynomial

Purpose: compute the equations of the isometry group of f.

Assume: f semiquasihomogeneous polynomial with an isolated singularity at 0

Return: list of two ring 'S1', 'S2'
- 'S1' contains the equations of the stabilizer (ideal 'stabid')
- 'S2' contains the action of the stabilizer (ideal 'actionid')

Global: varSubsList, contains the index j s.t. x(i) -> x(i)t(j) ...

Example:

```
LIB "qhmoduli.lib";
ring B = 0,(x,y,z), ls;
poly f = -z5+y5+x2z+x2y;
list stab = StabEqn(f);
def S1 = stab[1]; setring S1;  stabid;
↦ stabid[1]=s(2)*s(3)
↦ stabid[2]=s(1)^2*s(2)+s(1)^2*s(3)-1
↦ stabid[3]=s(1)^2*s(3)^2-s(3)
↦ stabid[4]=s(2)^4-s(3)^4+s(1)^2
↦ stabid[5]=s(1)^4+s(2)^3-s(3)^3
↦ stabid[6]=s(3)^5-s(1)^2*s(3)
def S2 = stab[2]; setring S2;  actionid;
↦ actionid[1]=s(1)*x
↦ actionid[2]=s(3)*y+s(2)*z
↦ actionid[3]=s(2)*y+s(3)*z
```

D.5.7.5 StabEqnId

Procedure from library `qhmoduli.lib` (see Section D.5.7 [qhmoduli_lib], page 476).

Usage: StabEqn(I, w); I ideal, w intvec

Purpose: compute the equations of the isometry group of the ideal I each generator of I is fixed by the stabilizer.

Assume: I semiquasihomogeneous ideal wrt 'w' with an isolated singularity at 0

Return: list of two ring 'S1', 'S2'
- 'S1' contains the equations of the stabilizer (ideal 'stabid')
- 'S2' contains the action of the stabilizer (ideal 'actionid')

Global: varSubsList, contains the index j s.t. t(i) -> t(i)t(j) ...

Example:

```
LIB "qhmoduli.lib";
ring B   = 0,(x,y,z), ls;
ideal I = x2,y3,z6;
intvec w = 3,2,1;
list stab = StabEqnId(I, w);
↦ // ** redefining d **
↦ // ** redefining newcoMx **
↦ // ** redefining coMx **
↦ // ** redefining d **
↦ // ** redefining newcoMx **
↦ // ** redefining coMx **
def S1 = stab[1]; setring S1;  stabid;
↦ stabid[1]=s(1)^2-1
↦ stabid[2]=s(2)^3-1
```

```
↦ stabid[3]=s(3)^6-1
def S2 = stab[2]; setring S2;  actionid;
↦ actionid[1]=s(1)*x
↦ actionid[2]=s(2)*y
↦ actionid[3]=s(3)*z
```

D.5.7.6 StabOrder

Procedure from library `qhmoduli.lib` (see Section D.5.7 [qhmoduli_lib], page 476).

Usage: StabOrder(f); poly f;

Purpose: compute the order of the stabilizer group of f.

Assume: f quasihomogeneous polynomial with an isolated singularity at 0

Return: int

Global: varSubsList

D.5.7.7 UpperMonomials

Procedure from library `qhmoduli.lib` (see Section D.5.7 [qhmoduli_lib], page 476).

Usage: UpperMonomials(poly f, [intvec w])

Purpose: compute the upper monomials of the milnor algebra of f.

Assume: f is quasihomogeneous (w.r.t. w)

Return: ideal

Example:

```
LIB "qhmoduli.lib";
ring B   = 0,(x,y,z), ls;
poly f = -z5+y5+x2z+x2y;
UpperMonomials(f);
↦ _[1]=y3z3
↦ _[2]=x2y3
↦ _[3]=x2y2
```

D.5.7.8 Max

Procedure from library `qhmoduli.lib` (see Section D.5.7 [qhmoduli_lib], page 476).

Usage: Max(data); intvec/list of integers data

Purpose: find the maximal integer contained in 'data'

Return: list

Assume: 'data' contains only integers and is not empty

D.5.7.9 Min

Procedure from library `qhmoduli.lib` (see Section D.5.7 [qhmoduli_lib], page 476).

Usage: Min(data); intvec/list of integers data

Purpose: find the minimal integer contained in 'data'

Return: list

Assume: 'data' contains only integers and is not empty

D.5.7.10 Table

Procedure from library `qhmoduli.lib` (see Section D.5.7 [qhmoduli_lib], page 476).

Usage: Table(cmd,i, lb, ub); string cmd, i; int lb, ub

Purpose: generate a list of size ub - lb + 1 s.t. _[i] = cmd(i)

Return: list

D.5.8 sing_lib

Library: sing.lib

Purpose: Invariants of Singularities

Authors: Gert-Martin Greuel, email: greuel@mathematik.uni-kl.de
Bernd Martin, email: martin@math.tu-cottbus.de

Procedures:

D.5.8.1 deform

Procedure from library `sing.lib` (see Section D.5.8 [sing_lib], page 480).

Usage: deform(id); id=ideal or poly

Return: matrix, columns are kbase of infinitesimal deformations

Example:

```
LIB "sing.lib";
ring r   = 32003,(x,y,z),ds;
ideal i  = xy,xz,yz;
matrix T = deform(i);
print(T);
↦ x,0,0,
↦ 0,0,z,
↦ 0,y,0
print(deform(x3+y5+z2));
↦ xy3,y3,xy2,y2,xy,y,x,1
```

D.5.8.2 dim_slocus

Procedure from library `sing.lib` (see Section D.5.8 [sing_lib], page 480).

Usage: dim_slocus(i); i ideal or poly

Return: dimension of singular locus of i

Example:

```
LIB "sing.lib";
ring r  = 32003,(x,y,z),ds;
ideal i = x5+y6+z6,x2+2y2+3z2;
dim_slocus(i);
↦ 0
```

D.5.8.3 is_active

Procedure from library `sing.lib` (see Section D.5.8 [sing_lib], page 480).

Usage: is_active(f,id); f poly, id ideal or module

Return: 1 if f is an active element modulo id (i.e. dim(id)=dim(id+f*R^n)+1, if id is a submodule of R^n) resp. 0 if f is not active. The basering may be a quotient ring

Note: regular parameters are active but not vice versa (id may have embedded components). proc is_reg tests whether f is a regular parameter

Example:

```
LIB "sing.lib";
ring r   =32003,(x,y,z),ds;
ideal i  = yx3+y,yz3+y3z;
poly f   = x;
is_active(f,i);
↦ 1
qring q  = std(x4y5);
poly f   = x;
module m = [yx3+x,yx3+y3x];
is_active(f,m);
↦ 0
```

D.5.8.4 is_ci

Procedure from library `sing.lib` (see Section D.5.8 [sing_lib], page 480).

Usage: is_ci(i); i ideal

Return: intvec = sequence of dimensions of ideals (j[1],...,j[k]), for k=1,...,size(j), where j is minimal base of i. i is a complete intersection if last number equals nvars-size(i)

Note: dim(0-ideal) = -1. You may first apply simplify(i,10); in order to delete zeroes and multiples from set of generators
printlevel >=0: display comments (default)

Example:

```
LIB "sing.lib";
int p      = printlevel;
printlevel = 1;                // display comments
ring r     = 32003,(x,y,z),ds;
ideal i    = x4+y5+z6,xyz,yx2+xz2+zy7;
is_ci(i);
↦ // complete intersection of dim 0
↦ // dim-sequence:
↦ 2,1,0
i          = xy,yz;
is_ci(i);
↦ // no complete intersection
↦ // dim-sequence:
↦ 2,2
printlevel = p;
```

D.5.8.5 is_is

Procedure from library `sing.lib` (see Section D.5.8 [sing_lib], page 480).

Usage: is_is(id); id ideal or poly

Return: intvec = sequence of dimensions of singular loci of ideals generated by id[1]..id[i], k = 1..size(id);
dim(0-ideal) = -1;
id defines an isolated singularity if last number is 0

Note: printlevel >=0: display comments (default)

Example:

```
LIB "sing.lib";
int p      = printlevel;
printlevel = 1;
ring r     = 32003,(x,y,z),ds;
ideal i    = x2y,x4+y5+z6,yx2+xz2+zy7;
is_is(i);
↦ // dim of singular locus = 0
↦ // isolated singularity if last number is 0 in dim-sequence:
↦ 2,1,0
poly f     = xy+yz;
is_is(f);
↦ // dim of singular locus = 1
↦ // isolated singularity if last number is 0 in dim-sequence:
↦ 1
printlevel = p;
```

D.5.8.6 is_reg

Procedure from library `sing.lib` (see Section D.5.8 [sing_lib], page 480).

Usage: is_reg(f,id); f poly, id ideal or module

Return: 1 if multiplication with f is injective modulo id, 0 otherwise

Note: let R be the basering and id a submodule of R^n. The procedure checks injectivity of multiplication with f on R^n/id. The basering may be a quotient ring

Example:

```
LIB "sing.lib";
ring r  = 32003,(x,y),ds;
ideal i = x8,y8;
ideal j = (x+y)^4;
i       = intersect(i,j);
poly f  = xy;
is_reg(f,i);
↦ 0
```

D.5.8.7 is_regs

Procedure from library `sing.lib` (see Section D.5.8 [sing_lib], page 480).

Usage: is_regs(i[,id]); i poly, id ideal or module (default: id=0)

Return: 1 if generators of i are a regular sequence modulo id, 0 otherwise

Note: let R be the basering and id a submodule of R^n. The procedure checks injectivity of multiplication with i[k] on R^n/id+i[1..k-1]. The basering may be a quotient ring
printlevel >=0: display comments (default)
printlevel >=1: display comments during computation

Example:

```
LIB "sing.lib";
int p      = printlevel;
printlevel = 1;
ring r1    = 32003,(x,y,z),ds;
ideal i    = x8,y8,(x+y)^4;
is_regs(i);
↦ // checking whether element 1 is regular mod 1 .. 0
↦ // checking whether element 2 is regular mod 1 .. 1
↦ // checking whether element 3 is regular mod 1 .. 2
↦ // elements 1..2 are regular, 3 is not regular mod 1..2
↦ 0
module m   = [x,0,y];
i          = x8,(x+z)^4;;
is_regs(i,m);
↦ // checking whether element 1 is regular mod 1 .. 0
↦ // checking whether element 2 is regular mod 1 .. 1
↦ // elements are a regular sequence of length 2
↦ 1
printlevel = p;
```

D.5.8.8 milnor

Procedure from library `sing.lib` (see Section D.5.8 [sing_lib], page 480).

Usage: milnor(i); i ideal or poly

Return: Milnor number of i, if i is ICIS (isolated complete intersection singularity) in generic form, resp. -1 if not

Note: use proc nf_icis to put generators in generic form
printlevel >=0: display comments (default)

Example:

```
LIB "sing.lib";
int p      = printlevel;
printlevel = 1;
ring r     = 32003,(x,y,z),ds;
ideal j    = x5+y6+z6,x2+2y2+3z2,xyz+yx;
milnor(j);
↦ //sequence of discriminant numbers: 100,149,70
↦ 21
poly f     = x7+y7+(x-y)^2*x2y2+z2;
milnor(f);
↦ 28
printlevel = p;
```

D.5.8.9 nf_icis

Procedure from library `sing.lib` (see Section D.5.8 [sing_lib], page 480).

Usage: nf_icis(i); i ideal

Return: ideal = generic linear combination of generators of i if i is an ICIS (isolated complete intersection singularity), return i if not

Note: this proc is useful in connection with proc milnor
printlevel >=0: display comments (default)

Example:

```
LIB "sing.lib";
int p      = printlevel;
printlevel = 1;
ring r     = 32003,(x,y,z),ds;
ideal i    = x3+y4,z4+yx;
nf_icis(i);
↦ // complete intersection of dim 1
↦ // dim-sequence:
↦ // dim of singular locus = 0
↦ // isolated singularity if last number is 0 in dim-sequence:
↦ // dim of singular locus = 0
↦ // isolated singularity if last number is 0 in dim-sequence:
↦ // ICIS in generic form after 1 genericity loop(s)
↦ _[1]=2xy+x3+y4+2z4
↦ _[2]=xy+z4
ideal j    = x3+y4,xy,yz;
nf_icis(j);
↦ // no complete intersection
↦ // dim-sequence:
↦ // no complete intersection
↦ _[1]=x3+y4
↦ _[2]=xy
↦ _[3]=yz
printlevel = p;
```

D.5.8.10 slocus

Procedure from library `sing.lib` (see Section D.5.8 [sing_lib], page 480).

Usage: slocus(i); i ideal

Return: ideal of singular locus of i

Example:

```
LIB "sing.lib";
ring r  = 0,(u,v,w,x,y,z),dp;
ideal i = wx,wy,wz,vx,vy,vz,ux,uy,uz,y3-x2;;
slocus(i);
↦ _[1]=x
↦ _[2]=w
↦ _[3]=v
↦ _[4]=u
↦ _[5]=y2
```

D.5.8.11 qhspectrum

Procedure from library `sing.lib` (see Section D.5.8 [sing_lib], page 480).

Usage: qhspectrum(f,w); f=poly, w=intvec;

Assume: f is a weighted homogeneous isolated singularity w.r.t. the weights given by w; w must consist of as many positive integers as there are variables of the basering

Compute: the spectral numbers of the w-homogeneous polynomial f, computed in a ring of characteristic 0

Return: intvec d,s1,...,su where:
d = w-degree(f) and si/d = i-th spectral-number(f)
No return value if basering has parameters or if f is no isolated singularity, displays a warning in this case

Example:

```
LIB "sing.lib";
ring r;
poly f=x3+y5+z2;
intvec w=10,6,15;
qhspectrum(f,w);
↦ 30,1,7,11,13,17,19,23,29
// the spectrum numbers are:
// 1/30,7/30,11/30,13/30,17/30,19/30,23/30,29/30
```

D.5.8.12 Tjurina

Procedure from library `sing.lib` (see Section D.5.8 [sing_lib], page 480).

Usage: Tjurina(id[,<any>]); id=ideal or poly

Assume: id=ICIS (isolated complete intersection singularity)

Return: standard basis of Tjurina-module of id,
of type module if id=ideal, resp. of type ideal if id=poly. If a second argument is present (of any type) return a list:
[1] = Tjurina number,
[2] = k-basis of miniversal deformation,
[3] = SB of Tjurina module,
[4] = Tjurina module

Display: Tjurina number if printlevel >= 0 (default)

Note: Tjurina number = -1 implies that id is not an ICIS

Example:

```
LIB "sing.lib";
int p      = printlevel;
printlevel = 1;
ring r     = 0,(x,y,z),ds;
poly f     = x5+y6+z7+xyz;        // singularity T[5,6,7]
list T     = Tjurina(f,"");
↦ // Tjurina number = 16
show(T[1]);                       // Tjurina number, should be 16
↦ // int, size 1
↦ 16
show(T[2]);                       // basis of miniversal deformation
↦ // ideal, 16 generator(s)
↦ z6,
```

```
↦ z5,
↦ z4,
↦ z3,
↦ z2,
↦ z,
↦ y5,
↦ y4,
↦ y3,
↦ y2,
↦ y,
↦ x4,
↦ x3,
↦ x2,
↦ x,
↦ 1
show(T[3]);                        // SB of Tjurina ideal
↦ // ideal, 6 generator(s)
↦ xy+7z6,
↦ xz+6y5,
↦ yz+5x4,
↦ 5x5-6y6,
↦ 6y6,
↦ z7
show(T[4]); "";                    // Tjurina ideal
↦ // ideal, 4 generator(s)
↦ yz+5x4,
↦ xz+6y5,
↦ xy+7z6,
↦ xyz+x5+y6+z7
↦
ideal j    = x2+y2+z2,x2+2y2+3z2;
show(kbase(Tjurina(j)));           // basis of miniversal deformation
↦ // Tjurina number = 5
↦ // module, 5 generator(s)
↦ [z]
↦ [y]
↦ [x]
↦ [1]
↦ [0,1]
hilb(Tjurina(j));                  // Hilbert series of Tjurina module
↦ // Tjurina number = 5
↦ //         2 t^0
↦ //        -3 t^1
↦ //        -3 t^2
↦ //         7 t^3
↦ //        -3 t^4
↦
↦ //         2 t^0
↦ //         3 t^1
↦ // dimension    = 0
↦ // multiplicity = 5
printlevel = p;
```

D.5.8.13 tjurina

Procedure from library `sing.lib` (see Section D.5.8 [sing_lib], page 480).

Usage: tjurina(id); id=ideal or poly

Assume: id=ICIS (isolated complete intersection singularity)

Return: int = Tjurina number of id

Note: Tjurina number = -1 implies that id is not an ICIS

Example:

```
LIB "sing.lib";
ring r=32003,(x,y,z),(c,ds);
ideal j=x2+y2+z2,x2+2y2+3z2;
tjurina(j);
↦ 5
```

D.5.8.14 T_1

Procedure from library `sing.lib` (see Section D.5.8 [sing_lib], page 480).

Usage: T_1(id[,<any>]); id = ideal or poly

Return: T_1(id): of type module/ideal if id is of type ideal/poly. We call T_1(id) the T_1-module of id. It is a std basis of the presentation of 1st order deformations of P/id, if P is the basering. If a second argument is present (of any type) return a list of 3 modules:
[1]= T_1(id)
[2]= generators of normal bundle of id, lifted to P
[3]= module of relations of [2], lifted to P
(note: transpose[3]*[2]=0 mod id)
The list contains all non-easy objects which must be computed to get T_1(id).

Display: k-dimension of T_1(id) if printlevel >= 0 (default)

Note: T_1(id) itself is usually of minor importance. Nevertheless, from it all relevant information can be obtained. The most important are probably vdim(T_1(id)); (which computes the Tjurina number), hilb(T_1(id)); and kbase(T_1(id));
If T_1 is called with two argument, then matrix([2])*(kbase([1])) represents a basis of 1st order semiuniversal deformation of id (use proc 'deform', to get this in a direct way).
For a complete intersection the proc Tjurina is faster

Example:

```
LIB "sing.lib";
int p      = printlevel;
printlevel = 1;
ring r     = 32003,(x,y,z),(c,ds);
ideal i    = xy,xz,yz;
module T   = T_1(i);
↦ // dim T_1 = 3
vdim(T);                      // Tjurina number = dim_K(T_1), should be 3
↦ 3
list L=T_1(i,"");
↦ // dim T_1 = 3
```

```
module kB  = kbase(L[1]);
print(L[2]*kB);               // basis of 1st order miniversal deformation
↦ 0,0,0,
↦ z,0,0,
↦ 0,y,z
show(L[2]);                   // presentation of normal bundle
↦ // module, 6 generator(s)
↦ [x]
↦ [y,z]
↦ [0,x,y]
↦ [0,z]
↦ [0,0,y]
↦ [0,0,z]
print(L[3]);                  // relations of i
↦ z, 0,
↦ -y,y,
↦ 0, -x
print(transpose(L[3])*L[2]);  // should be 0 (mod i)
↦ xz,0, -xy,-yz,0,  0,
↦ 0, yz,0,  yz, -xy,-xz
printlevel = p;
```

D.5.8.15 T_2

Procedure from library `sing.lib` (see Section D.5.8 [sing_lib], page 480).

Usage: T_2(id[,<any>]); id = ideal

Return: T_2(id): T_2-module of id . This is a std basis of a presentation of the module of obstructions of R=P/id, if P is the basering. If a second argument is present (of any type) return a list of 4 modules and 1 ideal:
[1]= T_2(id)
[2]= standard basis of id (ideal)
[3]= module of relations of id (=1st syzygy module of id)
[4]= presentation of syz/kos
[5]= relations of Hom_P([3]/kos,R), lifted to P
The list contains all non-easy objects which must be computed to get T_2(id).

Display: k-dimension of T_2(id) if printlevel >= 0 (default)

Note: The most important information is probably vdim(T_2(id)). Use proc miniversal to get equations of miniversal deformation.

Example:

```
LIB "sing.lib";
int p      = printlevel;
printlevel = 1;
ring  r    = 32003,(x,y),(c,dp);
ideal j    = x6-y4,x6y6,x2y4-x5y2;
module T   = T_2(j);
↦ // dim T_2 = 6
vdim(T);
↦ 6
hilb(T);"";
↦ //         1 t^0
```

```
↦ //        -1 t^2
↦ //        -1 t^3
↦ //         1 t^5
↦
↦ //         1 t^0
↦ //         2 t^1
↦ //         2 t^2
↦ //         1 t^3
↦ // dimension   = 0
↦ // degree      = 0
↦
ring r1    = 0,(x,y,z),dp;
ideal id   = xy,xz,yz;
list L     = T_2(id,"");
↦ // dim T_2 = 0
vdim(L[1]);                          // vdim of T_2
↦ 0
print(L[3]);                         // syzygy module of id
↦ -z,-z,
↦ y, 0,
↦ 0, x
printlevel = p;
```

D.5.8.16 T_12

Procedure from library `sing.lib` (see Section D.5.8 [sing_lib], page 480).

Usage: T_12(i[,any]); i = ideal

Return: T_12(i): list of 2 modules:
* standard basis of T_1-module =T_1(i), 1st order deformations
* standard basis of T_2-module =T_2(i), obstructions of R=P/i
If a second argument is present (of any type) return a list of 9 modules, matrices, integers:
[1]= standard basis of T_1-module
[2]= standard basis of T_2-module
[3]= vdim of T_1
[4]= vdim of T_2
[5]= matrix, whose cols present infinitesimal deformations
[6]= matrix, whose cols are generators of relations of i(=syz(i))
[7]= matrix, presenting Hom_P(syz/kos,R), lifted to P
[8]= presentation of T_1-module, no std basis
[9]= presentation of T_2-module, no std basis

Display: k-dimension of T_1 and T_2 if printlevel >= 0 (default)

Note: Use proc miniversal from deform.lib to get miniversal deformation of i, the list contains all objects used by proc miniversal

Example:

```
LIB "sing.lib";
int p      = printlevel;
printlevel = 1;
ring r     = 200,(x,y,z,u,v),(c,ws(4,3,2,3,4));
ideal i    = xz-y2,yz2-xu,xv-yzu,yu-z3,z2u-yv,zv-u2;
```

```
//a cyclic quotient singularity
list L     = T_12(i,1);
↦ // dim T_1 = 5
↦ // dim T_2 = 3
print(L[5]);             //matrix of infin. deformations
↦ 0,  0,  0,  0,  0,
↦ yz, y,  z2, 0,  0,
↦ -z3,-z2,-zu,yz, yu,
↦ -z2,-z, -u, 0,  0,
↦ zu, u,  v,  -z2,-zu,
↦ 0,  0,  0,  u,  v
printlevel = p;
```

D.5.8.17 tangentcone

Procedure from library `sing.lib` (see Section D.5.8 [sing_lib], page 480).

Usage: tangentcone(id [,n]); id = ideal, n = int

Return: the tangent cone of id

Note: the procedure works for any monomial ordering.
If n=0 use std w.r.t. local ordering ds, if n=1 use locstd

Example:

```
LIB "sing.lib";
ring R = 0,(x,y,z),ds;
ideal i  = 7xyz+z5,x2+y3+z7,5z5+y5;
tangentcone(i);
↦ _[1]=x2
↦ _[2]=7xyz
↦ _[3]=y5+5z5
↦ _[4]=7y4z
↦ _[5]=35z6
```

D.5.9 spcurve_lib

Library: spcurve.lib

Purpose: Deformations and Invariants of CM-codim 2 Singularities

Author: Anne Fruehbis-Krueger, anne@mathematik.uni-kl.de

Procedures:

D.5.9.1 isCMcod2

Procedure from library `spcurve.lib` (see Section D.5.9 [spcurve_lib], page 490).

Usage: isCMcod2(i); i an ideal

Return: presentation matrix of i, if i is Cohen-Macaulay of codimension 2
a zero matrix otherwise

Example:

```
LIB "spcurve.lib";
ring r=32003,(x,y,z),ds;
ideal i=xz,yz,x^3-y^4;
print(isCMcod2(i));
↦ -y,-x2,
↦ x, y3,
↦ 0, z
```

D.5.9.2 CMtype

Procedure from library `spcurve.lib` (see Section D.5.9 [spcurve_lib], page 490).

Usage: CMtype(i); i an ideal, CM of codimension 2

Return: Cohen-Macaulay type of i (integer)
(-1, if i is not Cohen-Macaulay of codimension 2)

Example:

```
LIB "spcurve.lib";
ring r=32003,(x,y,z),ds;
ideal i=xy,xz,yz;
CMtype(i);
↦ 2
```

D.5.9.3 matrixT1

Procedure from library `spcurve.lib` (see Section D.5.9 [spcurve_lib], page 490).

Usage: matrixT1(M,n); M matrix, n integer

Assume: M is a presentation matrix of an ideal i, CM of codimension 2; consider i as a family of ideals in a ring in the first n variables where the remaining variables are considered as parameters

Return: list consisting of the k x (k+1) matrix M and a module K_M such that T1=Mat(k,k+1;R)/K_M is the space of first order deformations of i

Example:

```
LIB "spcurve.lib";
ring r=32003,(x(1),x(2),x(3)),ds;
ideal curve=x(1)*x(2),x(1)*x(3),x(2)*x(3);
matrix M=isCMcod2(curve);
matrixT1(M,3);
↦ [1]:
↦    _[1,1]=0
↦    _[1,2]=-x(3)
↦    _[2,1]=-x(2)
↦    _[2,2]=x(2)
↦    _[3,1]=x(1)
↦    _[3,2]=0
↦ [2]:
↦    _[1]=gen(5)
↦    _[2]=gen(4)-gen(3)
↦    _[3]=-gen(2)
↦    _[4]=x(1)*gen(5)-x(2)*gen(3)
↦    _[5]=x(1)*gen(6)-x(2)*gen(4)
```

```
↦     _[6]=x(2)*gen(3)-x(3)*gen(1)
↦     _[7]=x(2)*gen(4)-x(3)*gen(2)
↦     _[8]=-x(3)*gen(2)
↦     _[9]=x(2)*gen(2)-x(2)*gen(1)
↦     _[10]=x(1)*gen(1)
↦     _[11]=-x(3)*gen(4)
↦     _[12]=x(2)*gen(4)-x(2)*gen(3)
↦     _[13]=x(1)*gen(3)
↦     _[14]=-x(3)*gen(6)
↦     _[15]=x(2)*gen(6)-x(2)*gen(5)
↦     _[16]=x(1)*gen(5)
```

D.5.9.4 semiCMcod2

Procedure from library `spcurve.lib` (see Section D.5.9 [spcurve_lib], page 490).

Usage: semiCMcod2(M,t1); M matrix, t1 module

Assume: M is a presentation matrix of an ideal i, CM of codimension 2, and t1 is a presentation of the space of first order deformations of i ((M,t1) as returned by the procedure matrixT1)

Create: new basering with name rneu

Return: ideal in rneu describing the semiuniversal deformation of i

Note: The current basering should not contain any variables named A(j) where j is some integer!

Example:

```
LIB "spcurve.lib";
ring r=32003,(x(1),x(2),x(3)),ds;
ideal curve=x(1)*x(2),x(1)*x(3),x(2)*x(3);
matrix M=isCMcod2(curve);
list l=matrixT1(M,3);
semiCMcod2(l[1],std(l[2]));
↦ _[1]=A(2)*A(3)-x(2)*A(3)-x(1)*x(2)
↦ _[2]=A(1)*A(3)+x(1)*x(3)
↦ _[3]=-x(2)*A(1)-x(3)*A(2)+x(2)*x(3)
```

D.5.9.5 discr

Procedure from library `spcurve.lib` (see Section D.5.9 [spcurve_lib], page 490).

Usage: discr(sem,n); sem ideal, n integer

Assume: sem is the versal deformation of an ideal of codimension 2.
the first n variables of the ring are treated as variables all the others as parameters

Return: ideal describing the discriminant

Note: This is not a powerful algorithm!

Example:

```
LIB "spcurve.lib";
ring r=32003,(x(1),x(2),x(3)),ds;
ideal curve=x(1)*x(2),x(1)*x(3),x(2)*x(3);
matrix M=isCMcod2(curve);
```

```
list l=matrixT1(M,3);
def sem=semiCMcod2(l[1],std(l[2]));
basering;
↦ //   characteristic : 32003
↦ //   number of vars : 6
↦ //        block   1 : ordering ds
↦ //                  : names    x(1) x(2) x(3)
↦ //        block   2 : ordering dp
↦ //                  : names    A(1) A(2) A(3)
↦ //        block   3 : ordering C
discr(sem,3);
↦ _[1]=A(1)*A(2)*A(3)
```

D.5.9.6 qhmatrix

Procedure from library `spcurve.lib` (see Section D.5.9 [spcurve_lib], page 490).

Usage: qhmatrix(M); M a k x (k+1) matrix

Return: list, consisting of an integer vector containing the weights of the variables of the basering and an integer matrix giving the weights of the entries of M, if M is quasihomogeneous; zero integer vector and zero integer matrix, if M is not quasihomogeneous, i.e. does not allow row and column weights

Example:

```
LIB "spcurve.lib";
ring r=0,(x,y,z),ds;
matrix M[3][2]=z,0,y,x,x^3,y;
qhmatrix(M);
↦ [1]:
↦    1,2,1
↦ [2]:
↦    1,0,
↦    2,1,
↦    3,2
pmat(M);
↦ z,  0,
↦ y,  x,
↦ x3, y
```

D.5.9.7 relweight

Procedure from library `spcurve.lib` (see Section D.5.9 [spcurve_lib], page 490).

Assume: N is a non-zero matrix
W is an integer matrix of the same size as N
a is an integer vector giving the weights of the variables

Return: integer, max(a-weighted order(N_ij) - W_ij | all entries ij)
string "ERROR" if sizes do not match

Example:

```
LIB "spcurve.lib";
ring r=32003,(x,y,z),ds;
matrix N[2][3]=z,0,y,x,x^3,y;
```

```
intmat W[2][3]=1,1,1,1,1,1;
intvec a=1,1,1;
relweight(N,W,a);
↦ 2
```

D.5.9.8 posweight

Procedure from library `spcurve.lib` (see Section D.5.9 [spcurve_lib], page 490).

Usage: posweight(M,t1,n[,s]); M matrix, t1 module, n int, s string
n=0 : all deformations of non-negative weight
n=1 : only non-constant deformations of non-negative weight
n=2 : all deformations of positive weight
As an optional parameter the name of a new ring may be specified.

Assume: M is a presentation matrix of a Cohen-Macaulay codimension 2 ideal and t1 is its T1 space in matrix notation

Create: new basering (default name: rneu); a different name for this ring may be given as a 4th parameter

Return: list, consisting of a presentation matrix describing the deformation given by the generators of T1 of non-negative/positive weight and the weight vector for the new variables

Note: The current basering should not contain any variables named T(i) where i is some integer!

Example:

```
LIB "spcurve.lib";
ring r=32003,(x(1),x(2),x(3)),ds;
ideal curve=(x(3)-x(1)^2)*x(3),(x(3)-x(1)^2)*x(2),x(2)^2-x(1)^7*x(3);
matrix M=isCMcod2(curve);
list l=matrixT1(M,3);
list li=posweight(l[1],std(l[2]),0);
pmat(li[1]);
↦ T(2)+x(1)*T(1), -x(3)+x(1)^2,
↦ -x(3),          x(2),
↦ x(2),           -x(1)^7
li[2];
↦ 3,1
```

D.5.9.9 KSpencerKernel

Procedure from library `spcurve.lib` (see Section D.5.9 [spcurve_lib], page 490).

Usage: KSpencerKernel(M[,s][,v]); M matrix, s string, v intvec
optional parameters (please specify in this order, if both are present):
* s = first of the names of the new rings
e.g. "R" leads to ring names R and R1
* v of size n(n+1) leads to the following module ordering
gen(v[1]) > gen(v[2]) > ... > gen(v[n(n+1)]) where the matrix entry ij corresponds to gen((i-1)*n+j)

Assume: M is a quasihomogeneous n x (n+1) matrix where the n minors define an isolated space curve singularity

Create: 2 new rings (default names: rneu and reneu)
different ring names may be specified as a 2nd parameter

Return: coefficient matrix representing the kernel of the Kodaira- Spencer map of the family of non-negative deformations having the given singularity as special fibre

Note: * the initial basering should not contain variables with name e(i) or T(i), since those variable names will internally be used by the script
* setting an intvec with 5 entries and name watchProgress shows the progress of the computations:
watchProgress[1]>0 => option(prot) in groebner commands
watchProgress[2]>0 => trace output for highcorner
watchProgress[3]>0 => output of deformed matrix
watchProgress[4]>0 => result of elimination step
watchProgress[4]>1 => trace output of multiplications with xyz and subsequent reductions
watchProgress[5]>0 => matrix representing the kernel using print

Example:

```
LIB "spcurve.lib";
ring r=0,(x,y,z),ds;
matrix M[3][2]=z-x^7,0,y^2,z,x^9,y;
def KS=KSpencerKernel(M,"ar");
print(KS);
↦ T(7),   0,      0,      0,     0,     0,     0,     0,
↦ KS[2,1],6*T(3), 3*T(7), 0,     0,     0,     0,     0,
↦ KS[3,1],KS[3,2],KS[3,3],6*T(3),3*T(7),0,     0,     0,
↦ 10*T(4),8*T(1), 7*T(5), 5*T(2),4*T(6),2*T(8),2*T(3),T(7)
nameof(basering);
↦ ar
basering;
↦ //   characteristic : 0
↦ //   number of vars : 17
↦ //        block   1 : ordering Ws
↦ //                  : names    e(1) e(2) e(3) e(4) e(5) e(6) x y z
↦ //                  : weights  -21 -10 -32 -21 -27 -16 3 16 21
↦ //        block   2 : ordering wp
↦ //                  : names    T(1) T(2) T(3) T(4) T(5) T(6) T(7) T(8)
↦ //                  : weights     8    5    2   10    7    4    1    2
↦ //        block   3 : ordering C
```

D.5.10 spectrum_lib

Library: spectrum.lib

Purpose: Singularity Spectrum for Nondegenerate Singularities

Author: S. Endrass

Procedures:

D.5.10.1 spectrumnd

Procedure from library `spectrum.lib` (see Section D.5.10 [spectrum_lib], page 495).

Usage: spectrumnd(f[,1]); poly f

Assume: basering has characteristic 0 and local ordering,
f has isolated singularity at 0 and nondegenerate principal part

Return:
list S:
ideal S[1]: spectral numbers in increasing order
intvec S[2]:
int S[2][i]: multiplicity of spectral number S[1][i]

Note: if a second argument 1 is given,
no test for a degenerate principal part will be done
SEE_ALSO: gaussman_lib

Example:

```
LIB "spectrum.lib";
ring R=0,(x,y),ds;
poly f=x^31+x^6*y^7+x^2*y^12+x^13*y^2+y^29;
spectrumnd(f);
↦ [1]:
↦    _[1]=-67/79
↦    _[2]=-62/79
↦    _[3]=-45/58
↦    _[4]=-57/79
↦    _[5]=-41/58
↦    _[6]=-55/79
↦    _[7]=-52/79
↦    _[8]=-37/58
↦    _[9]=-50/79
↦    _[10]=-18/29
↦    _[11]=-47/79
↦    _[12]=-45/79
↦    _[13]=-33/58
↦    _[14]=-16/29
↦    _[15]=-43/79
↦    _[16]=-42/79
↦    _[17]=-40/79
↦    _[18]=-1/2
↦    _[19]=-15/31
↦    _[20]=-14/29
↦    _[21]=-38/79
↦    _[22]=-27/58
↦    _[23]=-14/31
↦    _[24]=-35/79
↦    _[25]=-25/58
↦    _[26]=-13/31
↦    _[27]=-33/79
↦    _[28]=-12/29
↦    _[29]=-23/58
↦    _[30]=-31/79
↦    _[31]=-12/31
↦    _[32]=-30/79
↦    _[33]=-21/58
↦    _[34]=-11/31
↦    _[35]=-28/79
```

```
↦     _[36]=-10/29
↦     _[37]=-26/79
↦     _[38]=-19/58
↦     _[39]=-10/31
↦     _[40]=-25/79
↦     _[41]=-9/29
↦     _[42]=-17/58
↦     _[43]=-23/79
↦     _[44]=-9/31
↦     _[45]=-8/29
↦     _[46]=-21/79
↦     _[47]=-15/58
↦     _[48]=-8/31
↦     _[49]=-20/79
↦     _[50]=-7/29
↦     _[51]=-19/79
↦     _[52]=-18/79
↦     _[53]=-7/31
↦     _[54]=-13/58
↦     _[55]=-6/29
↦     _[56]=-16/79
↦     _[57]=-6/31
↦     _[58]=-15/79
↦     _[59]=-11/58
↦     _[60]=-14/79
↦     _[61]=-5/29
↦     _[62]=-13/79
↦     _[63]=-5/31
↦     _[64]=-9/58
↦     _[65]=-11/79
↦     _[66]=-4/29
↦     _[67]=-4/31
↦     _[68]=-10/79
↦     _[69]=-7/58
↦     _[70]=-9/79
↦     _[71]=-3/29
↦     _[72]=-8/79
↦     _[73]=-3/31
↦     _[74]=-7/79
↦     _[75]=-5/58
↦     _[76]=-6/79
↦     _[77]=-2/29
↦     _[78]=-2/31
↦     _[79]=-5/79
↦     _[80]=-3/58
↦     _[81]=-4/79
↦     _[82]=-3/79
↦     _[83]=-1/29
↦     _[84]=-1/31
↦     _[85]=-2/79
↦     _[86]=-1/58
↦     _[87]=-1/79
↦     _[88]=0
```

```
↦    _[89]=1/79
↦    _[90]=1/58
↦    _[91]=2/79
↦    _[92]=1/31
↦    _[93]=1/29
↦    _[94]=3/79
↦    _[95]=4/79
↦    _[96]=3/58
↦    _[97]=5/79
↦    _[98]=2/31
↦    _[99]=2/29
↦    _[100]=6/79
↦    _[101]=5/58
↦    _[102]=7/79
↦    _[103]=3/31
↦    _[104]=8/79
↦    _[105]=3/29
↦    _[106]=9/79
↦    _[107]=7/58
↦    _[108]=10/79
↦    _[109]=4/31
↦    _[110]=4/29
↦    _[111]=11/79
↦    _[112]=9/58
↦    _[113]=5/31
↦    _[114]=13/79
↦    _[115]=5/29
↦    _[116]=14/79
↦    _[117]=11/58
↦    _[118]=15/79
↦    _[119]=6/31
↦    _[120]=16/79
↦    _[121]=6/29
↦    _[122]=13/58
↦    _[123]=7/31
↦    _[124]=18/79
↦    _[125]=19/79
↦    _[126]=7/29
↦    _[127]=20/79
↦    _[128]=8/31
↦    _[129]=15/58
↦    _[130]=21/79
↦    _[131]=8/29
↦    _[132]=9/31
↦    _[133]=23/79
↦    _[134]=17/58
↦    _[135]=9/29
↦    _[136]=25/79
↦    _[137]=10/31
↦    _[138]=19/58
↦    _[139]=26/79
↦    _[140]=10/29
↦    _[141]=28/79
```

```
↦     _[142]=11/31
↦     _[143]=21/58
↦     _[144]=30/79
↦     _[145]=12/31
↦     _[146]=31/79
↦     _[147]=23/58
↦     _[148]=12/29
↦     _[149]=33/79
↦     _[150]=13/31
↦     _[151]=25/58
↦     _[152]=35/79
↦     _[153]=14/31
↦     _[154]=27/58
↦     _[155]=38/79
↦     _[156]=14/29
↦     _[157]=15/31
↦     _[158]=1/2
↦     _[159]=40/79
↦     _[160]=42/79
↦     _[161]=43/79
↦     _[162]=16/29
↦     _[163]=33/58
↦     _[164]=45/79
↦     _[165]=47/79
↦     _[166]=18/29
↦     _[167]=50/79
↦     _[168]=37/58
↦     _[169]=52/79
↦     _[170]=55/79
↦     _[171]=41/58
↦     _[172]=57/79
↦     _[173]=45/58
↦     _[174]=62/79
↦     _[175]=67/79
↦  [2]:
↦     1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,2,1,1,1,1,1,1,2,1,1,1,1,1,1,\
   1,1,2,1,1,1,1,1,1,1,1,2,1,1,1,1,1,1,1,1,1,1,1,2,1,1,1,1,2,1,1,1,1,2,1,1,1\
   ,1,1,2,1,1,1,1,2,1,1,1,1,1,2,1,4,1,2,1,1,1,1,1,2,1,1,1,1,2,1,1,1,1,1,2,1,\
   1,1,1,2,1,1,1,1,2,1,1,1,1,1,1,1,1,1,1,1,2,1,1,1,1,1,1,1,1,2,1,1,1,1,1,1,1\
   ,1,2,1,1,1,1,1,1,2,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1
```

D.6 Invariant theory

D.6.1 finvar_lib

Library: finvar.lib

Purpose: Invariant Rings of Finite Groups

Author: Agnes E. Heydtmann, email: agnes@math.uni-sb.de

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

Main procedures: Auxiliary procedures:

D.6.1.1 invariant_ring

Procedure from library `finvar.lib` (see Section D.6.1 [finvar_lib], page 499).

Usage: invariant_ring(G1,G2,...[,flags]);
G1,G2,...: <matrices> generating a finite matrix group, flags: an optional <intvec> with three entries: if the first one equals 0, the program attempts to compute the Molien series and Reynolds operator, if it equals 1, the program is told that the Molien series should not be computed, if it equals -1 characteristic 0 is simulated, i.e. the Molien series is computed as if the base field were characteristic 0 (the user must choose a field of large prime characteristic, e.g. 32003) and if the first one is anything else, it means that the characteristic of the base field divides the group order (i.e. it will not even be attempted to compute the Reynolds operator or Molien series), the second component should give the size of intervals between canceling common factors in the expansion of Molien series, 0 (the default) means only once after generating all terms, in prime characteristic also a negative number can be given to indicate that common factors should always be canceled when the expansion is simple (the root of the extension field occurs not among the coefficients)

Return: primary and secondary invariants (both of type <matrix>) generating the invariant ring with respect to the matrix group generated by the matrices in the input and irreducible secondary invariants (type <matrix>) if the Molien series was available

Display: information about the various stages of the program if the third flag does not equal 0

Theory: 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 "Generating a Noetherian Normalization of the Invariant Ring of a Finite Group" by Decker, Heydtmann, Schreyer (1998)). In the
non-modular case secondary invariants are calculated by finding a basis (in terms of monomials) of the basering modulo the primary invariants, mapping to invariants with the Reynolds operator and using those or their power products such that they are linearly independent modulo the primary invariants (see "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).

Example:

```
LIB "finvar.lib";
ring R=0,(x,y,z),dp;
matrix A[3][3]=0,1,0,-1,0,0,0,0,-1;
matrix P,S,IS=invariant_ring(A);
print(P);
↦ z2,x2+y2,x2y2
print(S);
↦ 1,xyz,x2z-y2z,x3y-xy3
print(IS);
↦ xyz,x2z-y2z,x3y-xy3
```

D.6.1.2 invariant_ring_random

Procedure from library `finvar.lib` (see Section D.6.1 [finvar_lib], page 499).

Usage: invariant_ring_random(G1,G2,...,r[,flags]);
G1,G2,...: <matrices> generating a finite matrix group, r: an <int> where -|r| to |r| is the range of coefficients of random combinations of bases elements that serve as primary invariants, flags: an optional <intvec> with three entries: if the first equals 0, the program attempts to compute the Molien series and Reynolds operator, if it equals 1, the program is told that the Molien series should not be computed, if it equals -1 characteristic 0 is simulated, i.e. the Molien series is computed as if the base field were characteristic 0 (the user must choose a field of large prime characteristic, e.g. 32003) and if the first one is anything else, then the characteristic of the base field divides the group order (i.e. we will not even attempt to compute the Reynolds operator or Molien series), the second component should give the size of intervals between canceling common factors in the expansion of the Molien series, 0 (the default) means only once after generating all terms, in prime characteristic also a negative number can be given to indicate that common factors should always be canceled when the expansion is simple (the root of the extension field does not occur among the coefficients)

Return: primary and secondary invariants (both of type <matrix>) generating invariant ring with respect to the matrix group generated by the matrices in the input and irreducible secondary invariants (type <matrix>) if the Molien series was available

Display: information about the various stages of the program if the third flag does not equal 0

Theory: is the same as for invariant_ring except that random combinations of basis elements are chosen as candidates for primary invariants and hopefully they lower the dimension of the previously found primary invariants by the right amount.

Example:

```
LIB "finvar.lib";
ring R=0,(x,y,z),dp;
matrix A[3][3]=0,1,0,-1,0,0,0,0,-1;
matrix P,S,IS=invariant_ring_random(A,1);
print(P);
↦ z2,x2+y2,x4+y4-z4
print(S);
↦ 1,xyz,x2z-y2z,x3y-xy3
print(IS);
↦ xyz,x2z-y2z,x3y-xy3
```

D.6.1.3 primary_invariants

Procedure from library `finvar.lib` (see Section D.6.1 [finvar_lib], page 499).

Usage: primary_invariants(G1,G2,...[,flags]);
G1,G2,...: <matrices> generating a finite matrix group, flags: an optional <intvec> with three entries, if the first one equals 0 (also the default), the program attempts to compute the Molien series and Reynolds operator, if it equals 1, the program is told that the Molien series should not be computed, if it equals -1 characteristic 0 is simulated, i.e. the Molien series is computed as if the base field were characteristic 0 (the user must choose a field of large prime characteristic, e.g. 32003) and if the first one is anything else, it means that the characteristic of the base field divides the group order, the second component should give the size of intervals between canceling common factors in the expansion of the Molien series, 0 (the default) means only once after generating all terms, in prime characteristic also a negative number can be given

to indicate that common factors should always be canceled when the expansion is simple (the root of the extension field occurs not among the coefficients)

Display: information about the various stages of the program if the third flag does not equal 0

Return: primary invariants (type <matrix>) of the invariant ring and if computable Reynolds operator (type <matrix>) and Molien series (type <matrix>) or ring name (type string) where the Molien series can be found in the char p case; if the first flag is 1 and we are in the non-modular case then an <intvec> is returned giving some of the degrees where no non-trivial homogeneous invariants can be found

Theory: 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 (1998)).

Example:

```
LIB "finvar.lib";
ring R=0,(x,y,z),dp;
matrix A[3][3]=0,1,0,-1,0,0,0,0,-1;
list L=primary_invariants(A);
print(L[1]);
↦ z2,x2+y2,x2y2
```

D.6.1.4 primary_invariants_random

Procedure from library `finvar.lib` (see Section D.6.1 [finvar_lib], page 499).

Usage: primary_invariants_random(G1,G2,...,r[,flags]);
G1,G2,...: <matrices> generating a finite matrix group, r: an <int> where -|r| to |r| is the range of coefficients of the random combinations of bases elements, flags: an optional <intvec> with three entries, if the first one equals 0 (also the default), the program attempts to compute the Molien series and Reynolds operator, if it equals 1, the program is told that the Molien series should not be computed, if it equals -1 characteristic 0 is simulated, i.e. the Molien series is computed as if the base field were characteristic 0 (the user must choose a field of large prime characteristic, e.g. 32003) and if the first one is anything else, it means that the characteristic of the base field divides the group order, the second component should give the size of intervals between canceling common factors in the expansion of the Molien series, 0 (the default) means only once after generating all terms, in prime characteristic also a negative number can be given to indicate that common factors should always be canceled when the expansion is simple (the root of the extension field does not occur among the coefficients)

Display: information about the various stages of the program if the third flag does not equal 0

Return: primary invariants (type <matrix>) of the invariant ring and if computable Reynolds operator (type <matrix>) and Molien series (type <matrix>), if the first flag is 1 and we are in the non-modular case then an <intvec> is returned giving some of the degrees where no non-trivial homogeneous invariants can be found

Theory: Bases of homogeneous invariants are generated successively and random linear combinations are chosen as primary invariants that lower the dimension of the ideal generated by the previously found invariants (see "Generating a Noetherian Normalization of the Invariant Ring of a Finite Group" by Decker, Heydtmann, Schreyer (1998)).

Example:

```
LIB "finvar.lib";
ring R=0,(x,y,z),dp;
matrix A[3][3]=0,1,0,-1,0,0,0,0,-1;
list L=primary_invariants_random(A,1);
print(L[1]);
↦ z2,x2+y2,x4+y4-z4
```

D.6.1.5 cyclotomic

Procedure from library `finvar.lib` (see Section D.6.1 [finvar_lib], page 499).

Usage: cyclotomic(i); i integer > 0

Returns: the i-th cyclotomic polynomial (type <poly>) as one in the first ring variable

Theory: x^i-1 is divided by the j-th cyclotomic polynomial where j takes on the value of proper divisors of i

Example:

```
LIB "finvar.lib";
ring R=0,(x,y,z),dp;
print(cyclotomic(25));
↦ x20+x15+x10+x5+1
```

D.6.1.6 group_reynolds

Procedure from library `finvar.lib` (see Section D.6.1 [finvar_lib], page 499).

Usage: group_reynolds(G1,G2,...[,v]);
G1,G2,...: nxn <matrices> generating a finite matrix group, v: an optional <int>

Assume: n is the number of variables of the basering, g the number of group elements

Return: a <list>, the first list element will be a gxn <matrix> representing the Reynolds operator if we are in the non-modular case; if the characteristic is >0, minpoly==0 and the finite group non-cyclic the second list element is an <int> giving the lowest common multiple of the matrix group elements' order (used in molien); in general all other list elements are nxn <matrices> listing all elements of the finite group

Display: information if v does not equal 0

Theory: The entire matrix group is generated by getting all left products of generators with the new elements from the last run through the loop (or the generators themselves during the first run). All the ones that have been generated before are thrown out and the program terminates when no new elements found in one run. Additionally each time a new group element is found the corresponding ring mapping of which the Reynolds operator is made up is generated. They are stored in the rows of the first return value.

Example:

```
LIB "finvar.lib";
ring R=0,(x,y,z),dp;
matrix A[3][3]=0,1,0,-1,0,0,0,0,-1;
list L=group_reynolds(A);
print(L[1]);
↦ y, -x,-z,
↦ -x,-y,z,
↦ -y,x, -z,
```

```
↦ x, y, z
print(L[2..size(L)]);
↦ 0, 1,0,
↦ -1,0,0,
↦ 0, 0,-1
↦ -1,0, 0,
↦ 0, -1,0,
↦ 0, 0, 1
↦ 0,-1,0,
↦ 1,0, 0,
↦ 0,0, -1
↦ 1,0,0,
↦ 0,1,0,
↦ 0,0,1
```

D.6.1.7 molien

Procedure from library `finvar.lib` (see Section D.6.1 [finvar_lib], page 499).

Usage: molien(G1,G2,...[,ringname,lcm,flags]);
G1,G2,...: nxn <matrices>, all elements of a finite matrix group, ringname: a <string> giving a name for a new ring of characteristic 0 for the Molien series in case of prime characteristic, lcm: an <int> giving the lowest common multiple of the elements' orders in case of prime characteristic, minpoly==0 and a non-cyclic group, flags: an optional <intvec> with three components: if the first element is not equal to 0 characteristic 0 is simulated, i.e. the Molien series is computed as if the base field were characteristic 0 (the user must choose a field of large prime characteristic, e.g. 32003), the second component should give the size of intervals between canceling common factors in the expansion of the Molien series, 0 (the default) means only once after generating all terms, in prime characteristic also a negative number can be given to indicate that common factors should always be canceled when the expansion is simple (the root of the extension field does not occur among the coefficients)

Assume: n is the number of variables of the basering, G1,G2... are the group elements generated by group_reynolds(), lcm is the second return value of group_reynolds()

Return: in case of characteristic 0 a 1x2 <matrix> giving enumerator and denominator of Molien series; in case of prime characteristic a ring with the name 'ringname' of characteristic 0 is created where the same Molien series (named M) is stored

Display: information if the third component of flags does not equal 0

Theory: In characteristic 0 the terms 1/det(1-xE) for all group elements of the Molien series are computed in a straight forward way. In prime characteristic a Brauer lift is involved. The returned matrix gives enumerator and denominator of the expanded version where common factors have been canceled.

Example:

```
LIB "finvar.lib";
"         note the case of prime characteristic";
↦           note the case of prime characteristic
ring R=0,(x,y,z),dp;
matrix A[3][3]=0,1,0,-1,0,0,0,0,-1;
list L=group_reynolds(A);
matrix M=molien(L[2..size(L)]);
```

```
print(M);
↦ x3+x2-x+1,-x7+x6+x5-x4+x3-x2-x+1
ring S=3,(x,y,z),dp;
string newring="alksdfjlaskdjf";
matrix A[3][3]=0,1,0,-1,0,0,0,0,-1;
list L=group_reynolds(A);
molien(L[2..size(L)],newring);
setring alksdfjlaskdjf;
print(M);
↦ x3+x2-x+1,-x7+x6+x5-x4+x3-x2-x+1
setring S;
kill alksdfjlaskdjf;
```

D.6.1.8 reynolds_molien

Procedure from library `finvar.lib` (see Section D.6.1 [finvar_lib], page 499).

Usage: reynolds_molien(G1,G2,...[,ringname,flags]);
G1,G2,...: nxn <matrices> generating a finite matrix group, ringname: a <string> giving a name for a new ring of characteristic 0 for the Molien series in case of prime characteristic, flags: an optional <intvec> with three components: if the first element is not equal to 0 characteristic 0 is simulated, i.e. the Molien series is computed as if the base field were characteristic 0 (the user must choose a field of large prime characteristic, e.g. 32003) the second component should give the size of intervals between canceling common factors in the expansion of the Molien series, 0 (the default) means only once after generating all terms, in prime characteristic also a negative number can be given to indicate that common factors should always be canceled when the expansion is simple (the root of the extension field does not occur among the coefficients)

Assume: n is the number of variables of the basering, G1,G2... are the group elements generated by group_reynolds(), g is the size of the group

Return: a gxn <matrix> representing the Reynolds operator is the first return value and in case of characteristic 0 a 1x2 <matrix> giving enumerator and denominator of Molien series is the second one; in case of prime characteristic a ring with the name ‘ringname‘ of characteristic 0 is created where the same Molien series (named M) is stored

Display: information if the third component of flags does not equal 0

Theory: The entire matrix group is generated by getting all left products of the generators with new elements from the last run through the loop (or the generators themselves during the first run). All the ones that have been generated before are thrown out and the program terminates when are no new elements found in one run. Additionally each time a new group element is found the corresponding ring mapping of which the Reynolds operator is made up is generated. They are stored in the rows of the first return value. In characteristic 0 the terms 1/det(1-xE) is computed whenever a new element E is found. In prime characteristic a Brauer lift is involved and the terms are only computed after the entire matrix group is generated (to avoid the modular case). The returned matrix gives enumerator and denominator of the expanded version where common factors have been canceled.

Example:

```
LIB "finvar.lib";
"         note the case of prime characteristic";
```

```
↦          note the case of prime characteristic
ring R=0,(x,y,z),dp;
matrix A[3][3]=0,1,0,-1,0,0,0,0,-1;
matrix REY,M=reynolds_molien(A);
print(REY);
↦ y, -x,-z,
↦ -x,-y,z,
↦ -y,x, -z,
↦ x, y, z
print(M);
↦ x3+x2-x+1,-x7+x6+x5-x4+x3-x2-x+1
ring S=3,(x,y,z),dp;
string newring="Qadjoint";
matrix A[3][3]=0,1,0,-1,0,0,0,0,-1;
matrix REY=reynolds_molien(A,newring);
print(REY);
↦ y, -x,-z,
↦ -x,-y,z,
↦ -y,x, -z,
↦ x, y, z
setring Qadjoint;
print(M);
↦ x3+x2-x+1,-x7+x6+x5-x4+x3-x2-x+1
setring S;
kill Qadjoint;
```

D.6.1.9 partial_molien

Procedure from library `finvar.lib` (see Section D.6.1 [finvar_lib], page 499).

Usage: partial_molien(M,n[,p]);
M: a 1x2 <matrix>, n: an <int> indicating number of terms in the expansion, p: an optional <poly>

Assume: M is the return value of molien or the second return value of reynolds_molien, p ought to be the second return value of a previous run of partial_molien and avoids recalculating known terms

Return: n terms (type <poly>) of the partial expansion of the Molien series (first n if there is no third parameter given, otherwise the next n terms depending on a previous calculation) and an intermediate result (type <poly>) of the calculation to be used as third parameter in a next run of partial_molien

Theory: The following calculation is implemented:
(1+a1x+a2x^2+...+anx^n)/(1+b1x+b2x^2+...+bmx^m)=(1+(a1-b1)x+...
(1+b1x+b2x^2+...+bmx^m)
———————————
 (a1-b1)x+(a2-b2)x^2+...
 (a1-b1)x+b1(a1-b1)x^2+...

Example:

```
LIB "finvar.lib";
ring R=0,(x,y,z),dp;
matrix A[3][3]=0,1,0,-1,0,0,0,0,-1;
matrix REY,M=reynolds_molien(A);
```

```
poly p(1..2);
p(1..2)=partial_molien(M,5);
p(1);
↦ 4x5+5x4+2x3+2x2+1
p(1..2)=partial_molien(M,5,p(2));
p(1);
↦ 18x10+12x9+13x8+8x7+8x6
```

D.6.1.10 evaluate_reynolds

Procedure from library `finvar.lib` (see Section D.6.1 [finvar_lib], page 499).

Usage: evaluate_reynolds(REY,I);
REY: a <matrix> representing the Reynolds operator, I: an arbitrary <ideal>

Assume: REY is the first return value of group_reynolds() or reynolds_molien()

Returns: image of the polynomials defining I under the Reynolds operator (type <ideal>)

Note: the characteristic of the coefficient field of the polynomial ring should not divide the order of the finite matrix group

Theory: REY has been constructed in such a way that each row serves as a ring mapping of which the Reynolds operator is made up.

Example:

```
LIB "finvar.lib";
ring R=0,(x,y,z),dp;
matrix A[3][3]=0,1,0,-1,0,0,0,0,-1;
list L=group_reynolds(A);
ideal I=x2,y2,z2;
print(evaluate_reynolds(L[1],I));
↦ 1/2x2+1/2y2,
↦ 1/2x2+1/2y2,
↦ z2
```

D.6.1.11 invariant_basis

Procedure from library `finvar.lib` (see Section D.6.1 [finvar_lib], page 499).

Usage: invariant_basis(g,G1,G2,...);
g: an <int> indicating of which degree (>0) the homogeneous basis should be, G1,G2,...: <matrices> generating a finite matrix group

Returns: the basis (type <ideal>) of the space of invariants of degree g

Theory: A general polynomial of degree g is generated and the generators of the matrix group applied. The difference ought to be 0 and this way a system of linear equations is created. It is solved by computing syzygies.

Example:

```
LIB "finvar.lib";
ring R=0,(x,y,z),dp;
matrix A[3][3]=0,1,0,-1,0,0,0,0,-1;
print(invariant_basis(2,A));
↦ x2+y2,
↦ z2
```

D.6.1.12 invariant_basis_reynolds

Procedure from library `finvar.lib` (see Section D.6.1 [finvar_lib], page 499).

Usage: invariant_basis_reynolds(REY,d[,flags]);
REY: a <matrix> representing the Reynolds operator, d: an <int> indicating of which degree (>0) the homogeneous basis should be, flags: an optional <intvec> with two entries: its first component gives the dimension of the space (default <0 meaning unknown) and its second component is used as the number of polynomials that should be mapped to invariants during one call of evaluate_reynolds if the dimension of the space is unknown or the number such that number x dimension polynomials are mapped to invariants during one call of evaluate_reynolds

Assume: REY is the first return value of group_reynolds() or reynolds_molien() and flags[1] given by partial_molien

Return: the basis (type <ideal>) of the space of invariants of degree d

Theory: Monomials of degree d are mapped to invariants with the Reynolds operator. A linearly independent set is generated with the help of minbase.

Example:

```
LIB "finvar.lib";
ring R=0,(x,y,z),dp;
matrix A[3][3]=0,1,0,-1,0,0,0,0,-1;
intvec flags=0,1,0;
matrix REY,M=reynolds_molien(A,flags);
flags=8,6;
print(invariant_basis_reynolds(REY,6,flags));
↦ z6,
↦ x2z4+y2z4,
↦ x2y2z2,
↦ x3yz2-xy3z2,
↦ x4z2+y4z2,
↦ x4y2+x2y4,
↦ x5y-xy5,
↦ x6+y6
```

D.6.1.13 primary_char0

Procedure from library `finvar.lib` (see Section D.6.1 [finvar_lib], page 499).

Usage: primary_char0(REY,M[,v]);
REY: a <matrix> representing the Reynolds operator, M: a 1x2 <matrix> representing the Molien series, v: an optional <int>

Assume: REY is the first return value of group_reynolds or reynolds_molien and M the one of molien or the second one of reynolds_molien

Display: information about the various stages of the program if v does not equal 0

Return: primary invariants (type <matrix>) of the invariant ring

Theory: 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 (1998)).

Example:

```
LIB "finvar.lib";
ring R=0,(x,y,z),dp;
matrix A[3][3]=0,1,0,-1,0,0,0,0,-1;
matrix REY,M=reynolds_molien(A);
matrix P=primary_char0(REY,M);
print(P);
↦ z2,x2+y2,x2y2
```

D.6.1.14 primary_charp

Procedure from library `finvar.lib` (see Section D.6.1 [finvar_lib], page 499).

Usage: primary_charp(REY,ringname[,v]);
REY: a <matrix> representing the Reynolds operator, ringname: a <string> giving the name of a ring where the Molien series is stored, v: an optional <int>

Assume: REY is the first return value of group_reynolds or reynolds_molien and ringname gives the name of a ring of characteristic 0 that has been created by molien or reynolds_molien

Display: information about the various stages of the program if v does not equal 0

Return: primary invariants (type <matrix>) of the invariant ring

Theory: 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 (1998)).

Example:

```
LIB "finvar.lib";
ring R=3,(x,y,z),dp;
matrix A[3][3]=0,1,0,-1,0,0,0,0,-1;
list L=group_reynolds(A);
string newring="alskdfj";
molien(L[2..size(L)],newring);
matrix P=primary_charp(L[1],newring);
if(system("with","Namespaces")) { kill Top::`newring`; }
kill `newring`;
print(P);
↦ z2,x2+y2,x2y2
```

D.6.1.15 primary_char0_no_molien

Procedure from library `finvar.lib` (see Section D.6.1 [finvar_lib], page 499).

Usage: primary_char0_no_molien(REY[,v]);
REY: a <matrix> representing the Reynolds operator, v: an optional <int>

Assume: REY is the first return value of group_reynolds or reynolds_molien

Display: information about the various stages of the program if v does not equal 0

Return: primary invariants (type <matrix>) of the invariant ring and an <intvec> listing some of the degrees where no non-trivial homogeneous invariants are to be found

Theory: 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 (1998)).

Example:

```
LIB "finvar.lib";
ring R=0,(x,y,z),dp;
matrix A[3][3]=0,1,0,-1,0,0,0,0,-1;
list L=group_reynolds(A);
list l=primary_char0_no_molien(L[1]);
print(l[1]);
↦ z2,x2+y2,x2y2
```

D.6.1.16 primary_charp_no_molien

Procedure from library `finvar.lib` (see Section D.6.1 [finvar_lib], page 499).

Usage: primary_charp_no_molien(REY[,v]);
REY: a <matrix> representing the Reynolds operator, v: an optional <int>

Assume: REY is the first return value of group_reynolds or reynolds_molien

Display: information about the various stages of the program if v does not equal 0

Return: primary invariants (type <matrix>) of the invariant ring and an <intvec> listing some of the degrees where no non-trivial homogeneous invariants are to be found

Theory: 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 (1998)).

Example:

```
LIB "finvar.lib";
ring R=3,(x,y,z),dp;
matrix A[3][3]=0,1,0,-1,0,0,0,0,-1;
list L=group_reynolds(A);
list l=primary_charp_no_molien(L[1]);
print(l[1]);
↦ z2,x2+y2,x2y2
```

D.6.1.17 primary_charp_without

Procedure from library `finvar.lib` (see Section D.6.1 [finvar_lib], page 499).

Usage: primary_charp_without(G1,G2,...[,v]);
G1,G2,...: <matrices> generating a finite matrix group, v: an optional <int>

Display: information about the various stages of the program if v does not equal 0

Return: primary invariants (type <matrix>) of the invariant ring

Theory: 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 (1998)). No Reynolds operator or Molien series is used.

Example:

```
LIB "finvar.lib";
ring R=2,(x,y,z),dp;
matrix A[3][3]=0,1,0,-1,0,0,0,0,-1;
matrix P=primary_charp_without(A);
print(P);
↦ x+y,z,xy
```

D.6.1.18 primary_char0_random

Procedure from library `finvar.lib` (see Section D.6.1 [finvar_lib], page 499).

Usage: primary_char0_random(REY,M,r[,v]);
REY: a <matrix> representing the Reynolds operator, M: a 1x2 <matrix> representing the Molien series, r: an <int> where `-|r|` to `|r|` is the range of coefficients of the random combinations of bases elements, v: an optional <int>

Assume: REY is the first return value of group_reynolds or reynolds_molien and M the one of molien or the second one of reynolds_molien

Display: information about the various stages of the program if v does not equal 0

Return: primary invariants (type <matrix>) of the invariant ring

Theory: Bases of homogeneous invariants are generated successively and random linear combinations are chosen as primary invariants that lower the dimension of the ideal generated by the previously found invariants (see "Generating a Noetherian Normalization of the Invariant Ring of a Finite Group" by Decker, Heydtmann, Schreyer (1998)).

Example:

```
LIB "finvar.lib";
ring R=0,(x,y,z),dp;
matrix A[3][3]=0,1,0,-1,0,0,0,0,-1;
matrix REY,M=reynolds_molien(A);
matrix P=primary_char0_random(REY,M,1);
print(P);
↦ z2,x2+y2,x4+y4-z4
```

D.6.1.19 primary_charp_random

Procedure from library `finvar.lib` (see Section D.6.1 [finvar_lib], page 499).

Usage: primary_charp_random(REY,ringname,r[,v]);
REY: a <matrix> representing the Reynolds operator, ringname: a <string> giving the name of a ring where the Molien series is stored, r: an <int> where `-|r|` to `|r|` is the range of coefficients of the random combinations of bases elements, v: an optional <int>

Assume: REY is the first return value of group_reynolds or reynolds_molien and ringname gives the name of a ring of characteristic 0 that has been created by molien or reynolds_molien

Display: information about the various stages of the program if v does not equal 0

Return: primary invariants (type <matrix>) of the invariant ring

Theory: Bases of homogeneous invariants are generated successively and random linear combinations are chosen as primary invariants that lower the dimension of the ideal generated by the previously found invariants (see "Generating a Noetherian Normalization of the Invariant Ring of a Finite Group" by Decker, Heydtmann, Schreyer (1998)).

Example:

```
LIB "finvar.lib";
ring R=3,(x,y,z),dp;
matrix A[3][3]=0,1,0,-1,0,0,0,0,-1;
list L=group_reynolds(A);
string newring="alskdfj";
molien(L[2..size(L)],newring);
matrix P=primary_charp_random(L[1],newring,1);
if(system("with","Namespaces")) { kill Top::‘newring‘; }
kill ‘newring‘;
print(P);
↦ z2,x2+y2,x4+y4-z4
```

D.6.1.20 primary_char0_no_molien_random

Procedure from library `finvar.lib` (see Section D.6.1 [finvar_lib], page 499).

Usage: primary_char0_no_molien_random(REY,r[,v]);
REY: a <matrix> representing the Reynolds operator, r: an <int> where -|r| to |r| is the range of coefficients of the random combinations of bases elements, v: an optional <int>

Assume: REY is the first return value of group_reynolds or reynolds_molien

Display: information about the various stages of the program if v does not equal 0

Return: primary invariants (type <matrix>) of the invariant ring and an <intvec> listing some of the degrees where no non-trivial homogeneous invariants are to be found

Theory: Bases of homogeneous invariants are generated successively and random linear combinations are chosen as primary invariants that lower the dimension of the ideal generated by the previously found invariants (see "Generating a Noetherian Normalization of the Invariant Ring of a Finite Group" by Decker, Heydtmann, Schreyer (1998)).

Example:

```
LIB "finvar.lib";
ring R=0,(x,y,z),dp;
matrix A[3][3]=0,1,0,-1,0,0,0,0,-1;
list L=group_reynolds(A);
list l=primary_char0_no_molien_random(L[1],1);
print(l[1]);
↦ z2,x2+y2,x4+y4-z4
```

D.6.1.21 primary_charp_no_molien_random

Procedure from library `finvar.lib` (see Section D.6.1 [finvar_lib], page 499).

Usage: primary_charp_no_molien_random(REY,r[,v]);
REY: a <matrix> representing the Reynolds operator, r: an <int> where -|r| to |r| is the range of coefficients of the random combinations of bases elements, v: an optional <int>

Assume: REY is the first return value of group_reynolds or reynolds_molien

Display: information about the various stages of the program if v does not equal 0

Return: primary invariants (type <matrix>) of the invariant ring and an <intvec> listing some of the degrees where no non-trivial homogeneous invariants are to be found

Theory: Bases of homogeneous invariants are generated successively and random linear combinations are chosen as primary invariants that lower the dimension of the ideal generated by the previously found invariants (see "Generating a Noetherian Normalization of the Invariant Ring of a Finite Group" by Decker, Heydtmann, Schreyer (1998)).

Example:

```
LIB "finvar.lib";
ring R=3,(x,y,z),dp;
matrix A[3][3]=0,1,0,-1,0,0,0,0,-1;
list L=group_reynolds(A);
list l=primary_charp_no_molien_random(L[1],1);
print(l[1]);
↦ z2,x2+y2,x4+y4-z4
```

D.6.1.22 primary_charp_without_random

Procedure from library `finvar.lib` (see Section D.6.1 [finvar_lib], page 499).

Usage: primary_charp_without_random(G1,G2,...,r[,v]);
G1,G2,...: <matrices> generating a finite matrix group, r: an <int> where -|r| to |r| is the range of coefficients of the random combinations of bases elements, v: an optional <int>

Display: information about the various stages of the program if v does not equal 0

Return: primary invariants (type <matrix>) of the invariant ring

Theory: Bases of homogeneous invariants are generated successively and random linear combinations are chosen as primary invariants that lower the dimension of the ideal generated by the previously found invariants (see "Generating a Noetherian Normalization of the Invariant Ring of a Finite Group" by Decker, Heydtmann, Schreyer (1998)). No Reynolds operator or Molien series is used.

Example:

```
LIB "finvar.lib";
ring R=2,(x,y,z),dp;
matrix A[3][3]=0,1,0,-1,0,0,0,0,-1;
matrix P=primary_charp_without_random(A,1);
print(P);
↦ x+y,z,xy
```

D.6.1.23 power_products

Procedure from library `finvar.lib` (see Section D.6.1 [finvar_lib], page 499).

Usage: power_products(dv,d);
dv: an <intvec> giving the degrees of homogeneous polynomials, d: the degree of the desired power products

Return: a size(dv)*m <intmat> where each column ought to be interpreted as containing the exponents of the corresponding polynomials. The product of the powers is then homogeneous of degree d.

Example:

```
LIB "finvar.lib";
intvec dv=5,5,5,10,10;
print(power_products(dv,10));
↦     2     1     1     0     0     0     0     0
↦     0     1     0     2     1     0     0     0
↦     0     0     1     0     1     2     0     0
↦     0     0     0     0     0     0     1     0
↦     0     0     0     0     0     0     0     1
print(power_products(dv,7));
↦     0
↦     0
↦     0
↦     0
↦     0
```

D.6.1.24 secondary_char0

Procedure from library `finvar.lib` (see Section D.6.1 [finvar_lib], page 499).

Usage: secondary_char0(P,REY,M[,v]);
P: a 1xn <matrix> with primary invariants, REY: a gxn <matrix> representing the Reynolds operator, M: a 1x2 <matrix> giving numerator and denominator of the Molien series, v: an optional <int>

Assume: n is the number of variables of the basering, g the size of the group, REY is the 1st return value of group_reynolds(), reynolds_molien() or the second one of primary_invariants(), M the return value of molien() or the second one of reynolds_molien() or the third one of primary_invariants()

Return: secondary invariants of the invariant ring (type <matrix>) and irreducible secondary invariants (type <matrix>)

Display: information if v does not equal 0

Theory: The secondary invariants are calculated by finding a basis (in terms of monomials) of the basering modulo the primary invariants, mapping those to invariants with the Reynolds operator and using these images 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)).

Example:

```
LIB "finvar.lib";
ring R=0,(x,y,z),dp;
matrix A[3][3]=0,1,0,-1,0,0,0,0,-1;
list L=primary_invariants(A);
matrix S,IS=secondary_char0(L[1..3]);
print(S);
↦ 1,xyz,x2z-y2z,x3y-xy3
print(IS);
↦ xyz,x2z-y2z,x3y-xy3
```

D.6.1.25 secondary_charp

Procedure from library `finvar.lib` (see Section D.6.1 [finvar_lib], page 499).

Usage: secondary_charp(P,REY,ringname[,v]);
P: a 1xn <matrix> with primary invariants, REY: a gxn <matrix> representing the Reynolds operator, ringname: a <string> giving the name of a ring of characteristic 0 where the Molien series is stored, v: an optional <int>

Assume: n is the number of variables of the basering, g the size of the group, REY is the 1st return value of group_reynolds(), reynolds_molien() or the second one of primary_invariants(), ‘ringname‘ is a ring of char 0 that has been created by molien() or reynolds_molien() or primary_invariants()

Return: secondary invariants of the invariant ring (type <matrix>) and irreducible secondary invariants (type <matrix>)

Display: information if v does not equal 0

Theory: Secondary invariants are calculated by finding a basis (in terms of monomials) of the basering modulo primary invariants, mapping those to invariants with the Reynolds operator and using these images 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)).

Example:

```
LIB "finvar.lib";
ring R=3,(x,y,z),dp;
matrix A[3][3]=0,1,0,-1,0,0,0,0,-1;
list L=primary_invariants(A);
matrix S,IS=secondary_charp(L[1..size(L)]);
print(S);
↦ 1,xyz,x2z-y2z,x3y-xy3
print(IS);
↦ xyz,x2z-y2z,x3y-xy3
```

D.6.1.26 secondary_no_molien

Procedure from library `finvar.lib` (see Section D.6.1 [finvar_lib], page 499).

Usage: secondary_no_molien(P,REY[,deg_vec,v]);
P: a 1xn <matrix> with primary invariants, REY: a gxn <matrix> representing the Reynolds operator, deg_vec: an optional <intvec> listing some degrees where no non-trivial homogeneous invariants can be found, v: an optional <int>

Assume: n is the number of variables of the basering, g the size of the group, REY is the 1st return value of group_reynolds(), reynolds_molien() or the second one of primary_invariants(), deg_vec is the second return value of primary_char0_no_molien(), primary_charp_no_molien(), primary_char0_no_molien_random() or primary_charp_no_molien_random()

Return: secondary invariants of the invariant ring (type <matrix>)

Display: information if v does not equal 0

Theory: Secondary invariants are calculated by finding a basis (in terms of monomials) of the basering modulo primary invariants, mapping those to invariants with the Reynolds operator and using these images as candidates for secondary invariants.

Example:

```
LIB "finvar.lib";
ring R=0,(x,y,z),dp;
matrix A[3][3]=0,1,0,-1,0,0,0,0,-1;
list L=primary_invariants(A,intvec(1,1,0));
matrix S=secondary_no_molien(L[1..3]);
print(S);
↦ 1,xyz,x2z-y2z,x3y-xy3
```

D.6.1.27 secondary_and_irreducibles_no_molien

Procedure from library `finvar.lib` (see Section D.6.1 [finvar_lib], page 499).

Usage: secondary_and_irreducibles_no_molien(P,REY[,v]);
P: a 1xn <matrix> with primary invariants, REY: a gxn <matrix> representing the Reynolds operator, v: an optional <int>

Assume: n is the number of variables of the basering, g the size of the group, REY is the 1st return value of group_reynolds(), reynolds_molien() or the second one of primary_invariants()

Return: secondary invariants of the invariant ring (type <matrix>) and irreducible secondary invariants (type <matrix>)

Display: information if v does not equal 0

Theory: Secondary invariants are calculated by finding a basis (in terms of monomials) of the basering modulo primary invariants, mapping those to invariants with the Reynolds operator and using these images 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)).

Example:

```
LIB "finvar.lib";
ring R=0,(x,y,z),dp;
matrix A[3][3]=0,1,0,-1,0,0,0,0,-1;
list L=primary_invariants(A,intvec(1,1,0));
matrix S,IS=secondary_and_irreducibles_no_molien(L[1..2]);
print(S);
↦ 1,xyz,x2z-y2z,x3y-xy3
print(IS);
↦ xyz,x2z-y2z,x3y-xy3
```

D.6.1.28 secondary_not_cohen_macaulay

Procedure from library `finvar.lib` (see Section D.6.1 [finvar_lib], page 499).

Usage: secondary_not_cohen_macaulay(P,G1,G2,...[,v]);
P: a 1xn <matrix> with primary invariants, G1,G2,...: nxn <matrices> generating a finite matrix group, v: an optional <int>

Assume: n is the number of variables of the basering

Return: secondary invariants of the invariant ring (type <matrix>)

Display: information if v does not equal 0

Theory: Secondary invariants are generated following "Generating Invariant Rings of Finite Groups over Arbitrary Fields" by Kemper (1996).

Example:

```
LIB "finvar.lib";
ring R=2,(x,y,z),dp;
matrix A[3][3]=0,1,0,-1,0,0,0,0,-1;
list L=primary_invariants(A);
matrix S=secondary_not_cohen_macaulay(L[1],A);
print(S);
↦ 1
```

D.6.1.29 orbit_variety

Procedure from library `finvar.lib` (see Section D.6.1 [finvar_lib], page 499).

Usage: orbit_variety(F,s);
F: a 1xm <matrix> defining an invariant ring, s: a <string> giving the name for a new ring

Return: a Groebner basis (type <ideal>, named G) for the ideal defining the orbit variety (i.e. the syzygy ideal) in the new ring (named 's')

Theory: The ideal of algebraic relations of the invariant ring generators is calculated, then the variables of the original ring are eliminated and the polynomials that are left over define the orbit variety

Example:

```
LIB "finvar.lib";
ring R=0,(x,y,z),dp;
matrix F[1][7]=x2+y2,z2,x4+y4,1,x2z-1y2z,xyz,x3y-1xy3;
string newring="E";
orbit_variety(F,newring);
print(G);
↦ y(4)-1,
↦ y(5)*y(6)-y(2)*y(7),
↦ y(2)*y(3)-y(5)^2-2*y(6)^2,
↦ y(1)^2*y(6)-2*y(3)*y(6)+y(5)*y(7),
↦ y(1)^2*y(5)-y(3)*y(5)-2*y(6)*y(7),
↦ y(1)^2*y(2)-y(2)*y(3)-2*y(6)^2,
↦ y(1)^4-3*y(1)^2*y(3)+2*y(3)^2+2*y(7)^2
basering;
↦ //   characteristic : 0
↦ //   number of vars : 7
↦ //        block   1 : ordering dp
↦ //                  : names    y(1) y(2) y(3) y(4) y(5) y(6) y(7)
↦ //        block   2 : ordering C
```

D.6.1.30 relative_orbit_variety

Procedure from library `finvar.lib` (see Section D.6.1 [finvar_lib], page 499).

Usage: relative_orbit_variety(I,F,s);
I: an <ideal> invariant under the action of a group, F: a 1xm <matrix> defining the invariant ring of this group, s: a <string> giving a name for a new ring

Return: a Groebner basis (type <ideal>, named G) for the ideal defining the relative orbit variety with respect to I in the new ring (named s)

Theory: A Groebner basis of the ideal of algebraic relations of the invariant ring generators is calculated, then one of the basis elements plus the ideal generators. The variables of the original ring are eliminated and the polynomials that are left define the relative orbit variety with respect to I.

Example:

```
LIB "finvar.lib";
ring R=0,(x,y,z),dp;
matrix F[1][3]=x+y+z,xy+xz+yz,xyz;
ideal I=x2+y2+z2-1,x2y+y2z+z2x-2x-2y-2z,xy2+yz2+zx2-2x-2y-2z;
string newring="E";
relative_orbit_variety(I,F,newring);
print(G);
↦ 27*y(3)^6-513*y(3)^4+33849*y(3)^2-784,
↦ 1475*y(2)+9*y(3)^4-264*y(3)^2+736,
↦ 8260*y(1)+9*y(3)^5-87*y(3)^3+5515*y(3)
basering;
↦ //   characteristic : 0
↦ //   number of vars : 3
↦ //        block   1 : ordering lp
↦ //                  : names    y(1) y(2) y(3)
↦ //        block   2 : ordering C
```

D.6.1.31 image_of_variety

Procedure from library `finvar.lib` (see Section D.6.1 [finvar_lib], page 499).

Usage: image_of_variety(I,F);
I: an arbitrary <ideal>, F: a 1xm <matrix> defining an invariant ring of a some matrix group

Return: the <ideal> defining the image under that group of the variety defined by I

Theory: relative_orbit_variety(I,F,s) is called and the newly introduced variables in the output are replaced by the generators of the invariant ring. This ideal in the original variables defines the image of the variety defined by I

Example:

```
LIB "finvar.lib";
ring R=0,(x,y,z),dp;
matrix F[1][3]=x+y+z,xy+xz+yz,xyz;
ideal I=xy;
print(image_of_variety(I,F));
↦ xyz
```

D.6.2 ainvar_lib

Library: ainvar.lib

Purpose: Invariant Rings of the Additive Group

Authors: Gerhard Pfister (email: pfister@mathematik.uni-kl.de), Gert-Martin Greuel (email: greuel@mathematik.uni-kl.de)

Procedures:

D.6.2.1 invariantRing

Procedure from library `ainvar.lib` (see Section D.6.2 [ainvar_lib], page 518).

Usage: invariantRing(m,p,q,b[,r,pa]); m matrix, p,q poly, b,r int, pa string

Assume: p,q variables with m(p)=q and q invariant under m
i.e. if p=x(i) and q=x(j) then m[j,1]=0 and m[i,1]=x(j)

Return: ideal, containing generators of the ring of invariants of the additive group (K,+) given by the vector field

m = m[1,1]*d/dx(1) +...+ m[n,1]*d/dx(n).

If b>0 the computation stops after all invariants of degree <= b (and at least one of higher degree) are found or when all invariants are computed.
If b<=0, the computation continues until all generators of the ring of invariants are computed (should be used only if the ring of invariants is known to be finitely generated otherwise the algorithm might not stop).
If r=1 a different reduction is used which is sometimes faster (default r=0).

Display: if pa is given (any string as 5th or 6th argument), the computation pauses whenever new invariants are found and displays them

Theory: The algorithm to compute the ring of invariants works in char 0 or big enough characteristic. (K,+) acts as the exponential of the vector field defined by the matrix m. For background see G.-M. Greuel, G. Pfister, Geometric quotients of unipotent group actions, Proc. London Math. Soc. (3) 67, 75-105 (1993).

Example:

```
LIB "ainvar.lib";
//Winkelmann: free action but Spec(k[x(1),...,x(5)]) --> Spec(invariant ring)
//is not surjective
ring rw=0,(x(1..5)),dp;
matrix m[5][1];
m[3,1]=x(1);
m[4,1]=x(2);
m[5,1]=1+x(1)*x(4)+x(2)*x(3);
ideal in=invariantRing(m,x(3),x(1),0);      //compute full invarint ring
in;
↦ in[1]=x(1)
↦ in[2]=x(2)
↦ in[3]=x(2)*x(3)*x(4)-x(2)*x(5)+x(4)
↦ in[4]=x(1)*x(3)*x(4)-x(1)*x(5)+x(3)
//Deveney/Finston: The ring of invariants is not finitely generated
ring rf=0,(x(1..7)),dp;
matrix m[7][1];
m[4,1]=x(1)^3;
m[5,1]=x(2)^3;
m[6,1]=x(3)^3;
m[7,1]=(x(1)*x(2)*x(3))^2;
ideal in=invariantRing(m,x(4),x(1),6);      //all invariants up to degree 6
in;
↦ in[1]=x(1)
↦ in[2]=x(3)
↦ in[3]=x(2)
↦ in[4]=x(3)^3*x(4)-x(1)^3*x(6)
```

```
↦ in[5]=x(2)^3*x(4)-x(1)^3*x(5)
↦ in[6]=x(2)^2*x(3)^2*x(4)-x(1)*x(7)
↦ in[7]=x(1)^2*x(2)^2*x(6)-x(3)*x(7)
↦ in[8]=x(1)^2*x(3)^2*x(5)-x(2)*x(7)
↦ in[9]=x(1)^2*x(2)*x(3)^4*x(4)*x(5)+x(1)^2*x(2)^4*x(3)*x(4)*x(6)-x(1)^5*x(\
   2)*x(3)*x(5)*x(6)-2*x(2)^2*x(3)^2*x(4)*x(7)+x(1)*x(7)^2
```

D.6.2.2 derivate

Procedure from library `ainvar.lib` (see Section D.6.2 [ainvar_lib], page 518).

Usage: derivate(m,id); m matrix, id poly/vector/ideal

Assume: m is a nx1 matrix, where n = number of variables of the basering

Return: poly/vector/ideal (same type as input), result of applying the vector field by the matrix m componentwise to id;

Note: the vector field is m[1,1]*d/dx(1) +...+ m[1,n]*d/dx(n)

Example:

```
LIB "ainvar.lib";
ring q=0,(x,y,z,u,v,w),dp;
poly f=2xz-y2;
matrix m[6][1] =x,y,0,u,v;
derivate(m,f);
↦ -2y2+2xz
vector v = [2xz-y2,u6-3];
derivate(m,v);
↦ 6u6*gen(2)-2y2*gen(1)+2xz*gen(1)
derivate(m,ideal(2xz-y2,u6-3));
↦ _[1]=-2y2+2xz
↦ _[2]=6u6
```

D.6.2.3 actionIsProper

Procedure from library `ainvar.lib` (see Section D.6.2 [ainvar_lib], page 518).

Usage: actionIsProper(m); m matrix

Assume: m is a nx1 matrix, where n = number of variables of the basering

Return: int = 1, if the action defined by m is proper, 0 if not

Note: m defines a group action which is the exponential of the vector field m[1,1]*d/dx(1) +...+ m[1,n]*d/dx(n)

Example:

```
LIB "ainvar.lib";
ring rf=0,x(1..7),dp;
matrix m[7][1];
m[4,1]=x(1)^3;
m[5,1]=x(2)^3;
m[6,1]=x(3)^3;
m[7,1]=(x(1)*x(2)*x(3))^2;
actionIsProper(m);
↦ 0
```

```
ring rd=0,x(1..5),dp;
matrix m[5][1];
m[3,1]=x(1);
m[4,1]=x(2);
m[5,1]=1+x(1)*x(4)^2;
actionIsProper(m);
↦ 1
```

D.6.2.4 reduction

Procedure from library `ainvar.lib` (see Section D.6.2 [ainvar_lib], page 518).

Usage: reduction(p,I[,q,n]); p poly, I ideal, [q monomial, n int (optional)]

Return: a polynomial equal to p-H(f1,...,fr), in case the leading term LT(p) of p is of the form H(LT(f1),...,LT(fr)) for some polynomial H in r variables over the base field, I=f1,...,fr; if q is given, a maximal power a is computed such that q^a divides p-H(f1,...,fr), and then (p-H(f1,...,fr))/q^a is returned; return p if no H is found
if n=1, a different algorithm is chosen which is sometimes faster (default: n=0; q and n can be given (or not) in any order)

Note: this is a kind of SAGBI reduction in the subalgebra K[f1,...,fr] of the basering

Example:

```
LIB "ainvar.lib";
ring q=0,(x,y,z,u,v,w),dp;
poly p=x2yz-x2v;
ideal dom =x-w,u2w+1,yz-v;
reduction(p,dom);
↦ 2xyzw-yzw2-2xvw+vw2
reduction(p,dom,w);
↦ 2xyz-yzw-2xv+vw
```

D.6.2.5 completeReduction

Procedure from library `ainvar.lib` (see Section D.6.2 [ainvar_lib], page 518).

Usage: completeReduction(p,I[,q,n]); p poly, I ideal, [q monomial, n int]

Return: a polynomial, the SAGBI reduction of the polynomial p with I via the procedure 'reduction' as long as possible
if n=1, a different algorithm is chosen which is sometimes faster (default: n=0; q and n can be given (or not) in any order)

Note: help reduction; shows an explanation of SAGBI reduction

Example:

```
LIB "ainvar.lib";
ring q=0,(x,y,z,u,v,w),dp;
poly p=x2yz-x2v;
ideal dom =x-w,u2w+1,yz-v;
completeReduction(p,dom);
↦ 2xyzw-yzw2-2xvw+vw2
completeReduction(p,dom,w);
↦ 0
```

D.6.2.6 localInvar

Procedure from library `ainvar.lib` (see Section D.6.2 [ainvar_lib], page 518).

Usage: localInvar(m,p,q,h); m matrix, p,q,h polynomials

Assume: m(q) and h are invariant under the vector field m, i.e. m(m(q))=m(h)=0 h must be a ring variable

Return: a polynomial, the invariant polynomial of the vector field

m = m[1,1]*d/dx(1) +...+ m[n,1]*d/dx(n)

with respect to p,q,h. It is defined as follows: set inv = p if p is invariant, and else as inv = m(q)^N * sum_i=1..N-1{ (-1)^i*(1/i!)*m^i(p)*(q/m(q))^i } where m^N(p) = 0, m^(N-1)(p) != 0;
the result is inv divided by h as much as possible

Example:

```
LIB "ainvar.lib";
ring q=0,(x,y,z),dp;
matrix m[3][1];
m[2,1]=x;
m[3,1]=y;
poly in=localInvar(m,z,y,x);
in;
↦ -1/2y2+xz
```

D.6.2.7 furtherInvar

Procedure from library `ainvar.lib` (see Section D.6.2 [ainvar_lib], page 518).

Usage: furtherInvar(m,id,karl,q); m matrix, id,karl ideals, q poly, n int

Assume: karl,id,q are invariant under the vector field m,
moreover, q must be a variable

Return: list of two ideals, the first ideal contains further invariants of the vector field

m = sum m[i,1]*d/dx(i) with respect to id,p,q,

i.e. we compute elements in the (invariant) subring generated by id which are divisible by q and divide them by q as much as possible the second ideal contains all invariants given before if n=1, a different algorithm is chosen which is sometimes faster (default: n=0)

Example:

```
LIB "ainvar.lib";
ring r=0,(x,y,z,u),dp;
matrix m[4][1];
m[2,1]=x;
m[3,1]=y;
m[4,1]=z;
ideal id=localInvar(m,z,y,x),localInvar(m,u,y,x);
ideal karl=id,x;
list in=furtherInvar(m,id,karl,x);
in;
↦ [1]:
↦    _[1]=y2z2-8/3xz3-2y3u+6xyzu-3x2u2
```

```
↦ [2]:
↦    _[1]=-1/2y2+xz
↦    _[2]=1/3y3-xyz+x2u
↦    _[3]=x
```

D.6.2.8 sortier

Procedure from library `ainvar.lib` (see Section D.6.2 [ainvar_lib], page 518).

Usage: sortier(id); id ideal/module

Return: the same ideal/module but with generators ordered by there leading term, starting with the smallest

Example:

```
LIB "ainvar.lib";
ring q=0,(x,y,z,u,v,w),dp;
ideal i=w,x,z,y,v;
sortier(i);
↦ _[1]=w
↦ _[2]=v
↦ _[3]=z
↦ _[4]=y
↦ _[5]=x
```

D.6.3 rinvar_lib

Library: rinvar.lib

Purpose: Invariant Rings of Reductive Groups

Author: Thomas Bayer, tbayer@in.tum.de
http://wwwmayr.informatik.tu-muenchen.de/personen/bayert/ Current Adress: Institut fuer Informatik, TU Muenchen

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

Procedures: **See also:** Section D.5.7 [qhmoduli_lib], page 476; Section D.7.5 [zeroset_lib], page 562.

D.6.3.1 HilbertSeries

Procedure from library `rinvar.lib` (see Section D.6.3 [rinvar_lib], page 523).

Usage: HilbertSeries(I, w); ideal I, intvec wt

Purpose: compute the polynomial p of the Hilbert Series,represented by p/q, of the ring K[t_1,...,t_m,y_1,...,y_r]/I1 where 'w' are the weights of the variables, computed, e.g., by 'HilbertWeights', 'I1' is of the form I[1] - y_1,...,I[r] - y_r and is quasihomogeneous w.r.t. 'w'

Return: intvec

Note: the leading 0 of the result does not belong to p, but is needed in the hilbert-driven 'std'.

D.6.3.2 HilbertWeights

Procedure from library `rinvar.lib` (see Section D.6.3 [rinvar_lib], page 523).

Purpose: compute the weights of the "slack" variables needed for the computation of the algebraic relations of the generators of 'I' s.t. the Hilbert driven 'std' can be used.

Return: intvec

Assume: basering = K[t_1,...,t_m,...], 'I' is quasihomogeneous w.r.t. 'w' and contains only polynomials in t_1,...,t_m

D.6.3.3 ImageVariety

Procedure from library `rinvar.lib` (see Section D.6.3 [rinvar_lib], page 523).

Usage: ImageVariety(ideal I, F [, w]);ideal I; F is a list/ideal, intvec w.

Purpose: compute the Zariski closure of the image of the variety of I under the morphism F.

Note: if 'I' and 'F' are quasihomogeneous w.r.t. 'w' then the Hilbert-driven 'std' is used.

Return: polynomial ring over the same ground field, containing the ideal 'imageid'. The variables are Y(1),...,Y(k) where k = size(F) - 'imageid' is the ideal of the Zariski closure of F(X) where X is the variety of I.

Example:

```
LIB "rinvar.lib";
ring B   = 0,(x,y),dp;
ideal I  = x4 - y4;
ideal F  = x2, y2, x*y;
def R = ImageVariety(I, F);
setring R;
imageid;
↦ imageid[1]=Y(1)*Y(2)-Y(3)^2
↦ imageid[2]=Y(1)^2-Y(2)^2
↦ imageid[3]=Y(2)^3-Y(1)*Y(3)^2
```

D.6.3.4 ImageGroup

Procedure from library `rinvar.lib` (see Section D.6.3 [rinvar_lib], page 523).

Usage: ImageGroup(G, action); ideal G, action;

Purpose: compute the ideal of the image of G in GL(m,K) induced by the linear action 'action', where G is an algebraic group and 'action' defines an action of G on K^m (size(action) = m).

Return: ring, a polynomial ring over the same ground field as the basering, containing the ideals 'groupid' and 'actionid'.
- 'groupid' is the ideal of the image of G (order <= order of G) - 'actionid' defines the linear action of 'groupid' on K^m.

Note: 'action' and 'actionid' have the same orbits
all variables which give only rise to 0's in the m x m matrices of G have been omitted.

Assume: basering K[s(1..r),t(1..m)] has r + m variables, G is the ideal of an algebraic group and F is an action of G on K^m. G contains only the variables s(1)...s(r). The action 'action' is given by polynomials f_1,...,f_m in basering, s.t. on the ring level we have

K[t_1,...,t_m] –> K[s_1,...,s_r,t_1,...,t_m]/G
t_i –> f_i(s_1,...,s_r,t_1,...,t_m)

Example:

```
LIB "rinvar.lib";
ring B   = 0,(s(1..2), t(1..2)),dp;
ideal G = s(1)^3-1, s(2)^10-1;
ideal action = s(1)*s(2)^8*t(1), s(1)*s(2)^7*t(2);
def R = ImageGroup(G, action);
setring R;
groupid;
↦ groupid[1]=-s(1)+s(2)^4
↦ groupid[2]=s(1)^8-s(2)^2
↦ groupid[3]=s(1)^7*s(2)^2-1
actionid;
↦ actionid[1]=s(1)*t(1)
↦ actionid[2]=s(2)*t(2)
```

D.6.3.5 InvariantRing

Procedure from library `rinvar.lib` (see Section D.6.3 [rinvar_lib], page 523).

Usage: InvariantRing(G, Gact [, opt]); ideal G, Gact; int opt

Purpose: compute generators of the invariant ring of G w.r.t. the action 'Gact'

Assume: G is a finite group and 'Gact' is a linear action.

Return: polynomial ring over a simple extension of the ground field of the basering (the extension might be trivial), containing the ideals 'invars' and 'groupid' and the poly 'newA'
- 'invars' contains the algebra-generators of the invariant ring - 'groupid' is the ideal of G in the new ring
- 'newA' if the minpoly changes this is the new representation of the algebraic number, otherwise it is set to 'a'.

Note: the delivered ring might have a different minimal polynomial

Example:

```
LIB "rinvar.lib";
ring B = 0, (s(1..2), t(1..2)), dp;
ideal G = -s(1)+s(2)^3, s(1)^4-1;
ideal action = s(1)*t(1), s(2)*t(2);
def R = InvariantRing(std(G), action);
setring R;
invars;
↦ invars[1]=t(1)^4
↦ invars[2]=t(1)^3*t(2)^3
↦ invars[3]=t(1)^2*t(2)^6
↦ invars[4]=t(1)*t(2)^9
↦ invars[5]=t(2)^12
```

D.6.3.6 InvariantQ

Procedure from library `rinvar.lib` (see Section D.6.3 [rinvar_lib], page 523).

Usage: InvariantQ(f, G, action); poly f; ideal G, action

Purpose: check if the polynomial f is invariant w.r.t. G where G acts via 'action' on K^m.

Assume: basering = K[s_1,...,s_m,t_1,...,t_m] where K = Q of K = Q(a) and minpoly != 0, f contains only t_1,...,t_m, G is the ideal of an algebraic group and a standard basis.

Return: int;
0 if f is not invariant,
1 if f is invariant

Note: G need not be finite

D.6.3.7 LinearizeAction

Procedure from library `rinvar.lib` (see Section D.6.3 [rinvar_lib], page 523).

Usage: LinearizeAction(G,action,r); ideal G, action; int r

Purpose: linearize the group action 'action' and find an equivariant embedding of K^m where m = size(action).

Assume: G contains only variables var(1..r) (r = nrs)
basering = K[s(1..r),t(1..m)], K = Q or K = Q(a) and minpoly != 0.

Return: polynomial ring containing the ideals 'actionid', 'embedid', 'groupid' - 'actionid' is the ideal defining the linearized action of G - 'embedid' is a parameterization of an equivariant embedding (closed) - 'groupid' is the ideal of G in the new ring

Note: set printlevel > 0 to see a trace

Example:

```
LIB "rinvar.lib";
ring B   = 0,(s(1..5), t(1..3)),dp;
ideal G =  s(3)-s(4), s(2)-s(5), s(4)*s(5), s(1)^2*s(4)+s(1)^2*s(5)-1, s(1)^2*s(5)^2-s(5), s(4)^4-s(5)^4+s(1)^2, s(1)^4+s(4)^3-s(5)^3, s(5)^5-s(1)^2*s(5);
ideal action = -s(4)*t(1)+s(5)*t(1), -s(4)^2*t(2)+2*s(4)^2*t(3)^2+s(5)^2*t(2), s(4)*t(3)+s(5)*t(3);
LinearActionQ(action, 5);
↦ 0
def R = LinearizeAction(G, action, 5);
setring R;
R;
↦ //   characteristic : 0
↦ //   number of vars : 9
↦ //        block   1 : ordering dp
↦ //                  : names    s(1) s(2) s(3) s(4) s(5) t(1) t(2) t(3) t(\
   4)
↦ //        block   2 : ordering C
actionid;
↦ actionid[1]=-s(4)*t(1)+s(5)*t(1)
↦ actionid[2]=-s(4)^2*t(2)+s(5)^2*t(2)+2*s(4)^2*t(4)
↦ actionid[3]=s(4)*t(3)+s(5)*t(3)
↦ actionid[4]=s(4)^2*t(4)+s(5)^2*t(4)
embedid;
↦ embedid[1]=t(1)
↦ embedid[2]=t(2)
↦ embedid[3]=t(3)
↦ embedid[4]=t(3)^2
groupid;
↦ groupid[1]=s(3)-s(4)
```

```
↦ groupid[2]=s(2)-s(5)
↦ groupid[3]=s(4)*s(5)
↦ groupid[4]=s(1)^2*s(4)+s(1)^2*s(5)-1
↦ groupid[5]=s(1)^2*s(5)^2-s(5)
↦ groupid[6]=s(4)^4-s(5)^4+s(1)^2
↦ groupid[7]=s(1)^4+s(4)^3-s(5)^3
↦ groupid[8]=s(5)^5-s(1)^2*s(5)
LinearActionQ(actionid, 5);
↦ 1
```

D.6.3.8 LinearActionQ

Procedure from library `rinvar.lib` (see Section D.6.3 [rinvar_lib], page 523).

Usage: LinearActionQ(action,nrs,nrt); ideal action, int nrs

Purpose: check if the action defined by 'action' is linear w.r.t. the variables var(nrs + 1...nvars(basering)).

Return: 0 action not linear
1 action is linear

Example:

```
LIB "rinvar.lib";
ring R   = 0,(s(1..5), t(1..3)),dp;
ideal G =  s(3)-s(4), s(2)-s(5), s(4)*s(5), s(1)^2*s(4)+s(1)^2*s(5)-1, s(1)^2*s(5)^2-s(5), s(4)^4-s(5)^4+s(1)^2, s(1)^4+s(4)^3-s(5)^3, s(5)^5-s(1)^2*s(5);
ideal Gaction = -s(4)*t(1)+s(5)*t(1), -s(4)^2*t(2)+2*s(4)^2*t(3)^2+s(5)^2*t(2), s(4)*t(3)+s(5)*t(3);
LinearActionQ(Gaction, 5, 3);
↦ // ** too many arguments for LinearActionQ
↦ 0
```

D.6.3.9 LinearCombinationQ

Procedure from library `rinvar.lib` (see Section D.6.3 [rinvar_lib], page 523).

Usage: LinearCombination(I, f); ideal I, poly f

Purpose: test if f can be written as a linear combination of the generators of I.

Return: 0 f is not a linear combination
1 f is a linear combination

D.6.3.10 MinimalDecomposition

Procedure from library `rinvar.lib` (see Section D.6.3 [rinvar_lib], page 523).

Usage: MinimalDecomposition(f,a,b); poly f; int a, b.

Purpose: decompose f as a sum M[1,1]*M[2,1] + ... + M[1,r]*M[2,r] where M[1,i] contains only s(1..a), M[2,i] contains only t(1...b) s.t. r is minimal

Assume: f polynomial in K[s(1..a),t(1..b)], K = Q or K = Q(a) and minpoly != 0

Return: 2 x r matrix M s.t. f = M[1,1]*M[2,1] + ... + M[1,r]*M[2,r]

Example:

```
LIB "rinvar.lib";
ring R = 0, (s(1..2), t(1..2)), dp;
poly h = s(1)*(t(1) + t(1)^2) +  (t(2) + t(2)^2)*(s(1)^2 + s(2));
matrix M = MinimalDecomposition(h, 2, 2);
M;
↦ M[1,1]=s(1)^2+s(2)
↦ M[1,2]=s(1)
↦ M[2,1]=t(2)^2+t(2)
↦ M[2,2]=t(1)^2+t(1)
M[1,1]*M[2,1] + M[1,2]*M[2,2] - h;
↦ 0
```

D.6.3.11 NullCone

Procedure from library `rinvar.lib` (see Section D.6.3 [rinvar_lib], page 523).

Usage: NullCone(G, action); ideal G, action

Purpose: compute the ideal of the null cone of the linear action of G on K^n, given by 'action', by means of Derksen's algorithm

Assume: basering = K[s(1..r),t(1..n)], K = Q or K = Q(a) and minpoly != 0, G is an ideal of a reductive algebraic group in K[s(1..r)], 'action' is a linear group action of G on K^n (n = ncols(action))

Return: ideal of the null cone of G.

Note: the generators of the null cone are homogeneous, but i.g. not invariant

Example:

```
LIB "rinvar.lib";
ring R = 0, (s(1..2), x, y), dp;
ideal G = -s(1)+s(2)^3, s(1)^4-1;
ideal action = s(1)*x, s(2)*y;
ideal inv = NullCone(G, action);
inv;
↦ inv[1]=x^4
↦ inv[2]=x^3*y^3
↦ inv[3]=x^2*y^6
↦ inv[4]=x*y^9
↦ inv[5]=y^12
```

D.6.3.12 ReynoldsImage

Procedure from library `rinvar.lib` (see Section D.6.3 [rinvar_lib], page 523).

Usage: ReynoldsImage(RO, f); list RO, poly f

Purpose: compute the Reynolds image of the polynomial f where RO represents the Reynolds operator

Return: poly

D.6.3.13 ReynoldsOperator

Procedure from library `rinvar.lib` (see Section D.6.3 [rinvar_lib], page 523).

Usage: ReynoldsOperator(G, action [, opt); ideal G, action; int opt

Purpose: compute the Reynolds operator of the group G which act via 'action'

Return: polynomial ring R over a simple extension of the ground field of the basering (the extension might be trivial), containing a list 'ROelements', the ideals 'id', 'actionid' and the polynomial 'newA'. R = K(a)[s(1..r),t(1..n)].
- 'ROelements' is a list of ideal, each ideal represents a substitution map F : R -> R according to the zero-set of G - 'id' is the ideal of G in the new ring
- 'newA' is the new representation of a' in terms of a. If the basering does not contain a parameter then 'newA' = 'a'.

Assume: basering = K[s(1..r),t(1..n)], K = Q or K = Q(a') and minpoly != 0, G is the ideal of a finite group in K[s(1..r)], 'action' is a linear group action of G

D.6.3.14 SimplifyIdeal

Procedure from library `rinvar.lib` (see Section D.6.3 [rinvar_lib], page 523).

Purpose: simplify ideal I to the ideal I', do not change the names of the first m variables, new ideal I' might contain less variables. I' contains variables var(1..m)

Return: list
_[1] ideal I'
_[2] ideal representing a map phi to a ring with probably less vars. s.t. phi(I) = I'
_[3] list of variables
_[4] list from 'elimpart'

D.6.3.15 TransferIdeal

Procedure from library `rinvar.lib` (see Section D.6.3 [rinvar_lib], page 523).

D.6.4 stratify_lib

Library: stratify.lib

Purpose: Algorithmic Stratification for Unipotent Group-Actions

Author: Anne Fruehbis-Krueger, anne@mathematik.uni-kl.de

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

Procedures:

D.6.4.1 prepMat

Procedure from library `stratify.lib` (see Section D.6.4 [stratify_lib], page 529).

Usage: prepMat(M,wr,ws,step);
where M is a matrix, wr is an intvec of size ncols(M), ws an intvec of size nrows(M) and step is an integer

Return: 2 lists of submatrices corresponding to the filtrations specified by wr and ws
the first list corresponds to the list for the filtration of AdA, i.e. the ranks of these matrices will be the r_i, the second one to the list for the filtration of L, i.e. the ranks of these matrices will be the s_i

Note: * the entries of the matrix M are M_ij=delta_i(x_j),
* wr is used to determine what subset of the set of all dx_i is generating AdF^l(A):
if (k-1)*step <= wr[i] < k*step, then dx_i is in the set of generators of AdF^l(A) for all l>=k and the i-th column of M appears in each submatrix starting from the k-th
* ws is used to determine what subset of the set of all delta_i is generating Z_l(L):
if (k-1)*step <= ws[i] < k*step, then delta_i is in the set of generators of Z_l(A) for l < k and the i-th row of M appears in each submatrix up to the (k-1)th
* the entries of wr and ws as well as step should be positive integers

Example:

```
LIB "stratify.lib";
ring r=0,(t(1..3)),dp;
matrix M[2][3]=0,t(1),3*t(2),0,0,t(1);
print(M);
↦ 0,t(1),3*t(2),
↦ 0,0,   t(1)
intvec wr=1,3,5;
intvec ws=2,4;
int step=2;
prepMat(M,wr,ws,step);
↦ [1]:
↦    [1]:
↦       _[1,1]=0
↦       _[2,1]=0
↦    [2]:
↦       _[1,1]=0
↦       _[1,2]=t(1)
↦       _[2,1]=0
↦       _[2,2]=0
↦    [3]:
↦       _[1,1]=0
↦       _[1,2]=t(1)
↦       _[1,3]=3*t(2)
↦       _[2,1]=0
↦       _[2,2]=0
↦       _[2,3]=t(1)
↦ [2]:
↦    [1]:
↦       _[1,1]=0
↦       _[1,2]=t(1)
↦       _[1,3]=3*t(2)
↦       _[2,1]=0
↦       _[2,2]=0
↦       _[2,3]=t(1)
↦    [2]:
↦       _[1,1]=0
↦       _[1,2]=0
↦       _[1,3]=t(1)
```

D.6.4.2 stratify

Procedure from library `stratify.lib` (see Section D.6.4 [stratify_lib], page 529).

Usage: stratify(M,wr,ws,step);
where M is a matrix, wr is an intvec of size ncols(M), ws an intvec of size nrows(M) and step is an integer

Return: list of lists, each entry of the big list corresponds to one locally closed set and has the following entries:
1) intvec giving the corresponding rs-vector
2) ideal determining the closed set
3) list d of polynomials determining the open set D(d[1]) empty list if there is more than one open set
4-n) lists of polynomials determining open sets which all lead to the same rs-vector

Note: * ring ordering should be global, i.e. the ring should be a polynomial ring
* the entries of the matrix M are M_ij=delta_i(x_j),
* wr is used to determine what subset of the set of all dx_i is generating AdF^l(A):
if (k-1)*step < wr[i] <= k*step, then dx_i is in the set of generators of AdF^l(A) for all l>=k
* ws is used to determine what subset of the set of all delta_i is generating Z_l(L):
if (k-1)*step <= ws[i] < k*step, then delta_i is in the set of generators of Z_l(A) for l < k
* the entries of wr and ws as well as step should be positive integers
* the filtrations have to be known, no sanity checks concerning the filtrations are performed !!!

Example:

```
LIB "stratify.lib";
ring r=0,(t(1..3)),dp;
matrix M[2][3]=0,t(1),3*t(2),0,0,t(1);
intvec wr=1,3,5;
intvec ws=2,4;
int step=2;
stratify(M,wr,ws,step);
↦ [1]:
↦    [1]:
↦       0,0,0,0
↦    [2]:
↦       _[1]=t(2)
↦       _[2]=t(1)
↦    [3]:
↦       [1]:
↦          1
↦ [2]:
↦    [1]:
↦       0,1,0,1
↦    [2]:
↦       _[1]=t(1)
↦    [3]:
↦       [1]:
↦          t(2)
↦       [2]:
↦          t(2)
↦ [3]:
↦    [1]:
```

```
↦       1,2,1,2
↦    [2]:
↦       _[1]=0
↦    [3]:
↦       [1]:
↦          t(1)
↦       [2]:
↦          t(1)
```

D.7 Symbolic-numerical solving

D.7.1 presolve_lib

Library: presolve.lib

Purpose: Pre-Solving of Polynomial Equations

Author: Gert-Martin Greuel, email: greuel@mathematik.uni-kl.de,

Procedures:

D.7.1.1 degreepart

Procedure from library `presolve.lib` (see Section D.7.1 [presolve_lib], page 532).

Usage: degreepart(id,d1,d2[,v]); id=ideal/module, d1,d1=integers, v=intvec

Return: generators of id of [v-weighted] total degree >= d1 and <= d2 (default: v = 1,...,1)

Example:

```
LIB "presolve.lib";
ring r=0,(x,y,z),dp;
ideal i=1+x+x2+x3+x4,3,xz+y3+z8;
degreepart(i,0,4);
↦ _[1]=x4+x3+x2+x+1
↦ _[2]=3
module m=[x,y,z],x*[x3,y2,z],[1,x2,z3,0,1];
intvec v=2,3,6;
show(degreepart(m,8,8,v));
↦ // module, 1 generator(s)
↦ [x4,xy2,xz]
```

D.7.1.2 elimlinearpart

Procedure from library `presolve.lib` (see Section D.7.1 [presolve_lib], page 532).

Usage: elimlinearpart(i[,n]); i=ideal, n=integer,
default: n=nvars(basering)

Return: list L with 5 entries:
L[1]: (interreduced) ideal obtained from i by substituing
from the first n variables those, which appear in a linear part
of i, by putting this part into triangular form
L[2]: ideal of variables which have been substituted
L[3]: ideal, j-th element defines substitution of j-th var in [2]

L[4]: ideal of variables of basering, eliminated ones are set to 0
L[5]: ideal, describing the map from the basering to itself such that
L[1] is the image of i

Note: the procedure does always interreduce the ideal i internally w.r.t. ordering dp.

Example:

```
LIB "presolve.lib";
ring s=0,(x,y,z),dp;
ideal i = x3+y2+z,x2y2+z3,y+z+1;
elimlinearpart(i);
↦ [1]:
↦    _[1]=x3+z2+3z+1
↦    _[2]=x2z2+2x2z+z3+x2
↦ [2]:
↦    _[1]=y
↦ [3]:
↦    _[1]=y+z+1
↦ [4]:
↦    _[1]=x
↦    _[2]=0
↦    _[3]=z
↦ [5]:
↦    _[1]=x
↦    _[2]=-z-1
↦    _[3]=z
```

D.7.1.3 elimpart

Procedure from library `presolve.lib` (see Section D.7.1 [presolve_lib], page 532).

Usage: elimpart(i [,n,e]); i=ideal, n,e=integers
n : only the first n vars are considered for substitution,
e =0: substitute from linear part of i (same as elimlinearpart)
e!=0: eliminate also by direct substitution
(default: n = nvars(basering), e = 1)

Return: list of 5 objects:
[1]: ideal obtained by substituting from the first n variables those
from i, which appear in the linear part of i (or, if e!=0, which
can be expressed directly in the remaining vars)
[2]: ideal, variables which have been substituted
[3]: ideal, i-th element defines substitution of i-th var in [2]
[4]: ideal of variables of basering, substituted ones are set to 0
[5]: ideal, describing the map from the basering, say k[x(1..m)], to
itself onto k[..variables fom [4]..] and [1] is the image of i

The ideal i is generated by [1] and [3] in k[x(1..m)], the map [5] maps [3] to 0, hence induces an isomorphism

k[x(1..m)]/i -> k[..variables fom [4]..]/[1]

Note: If the basering has ordering (c,dp), this is faster for big ideals, since it avoids internal ring change and mapping.

Example:

```
LIB "presolve.lib";
ring s=0,(x,y,z),dp;
ideal i =x2+y2,x2+y+1;
elimpart(i,3,0);
↦ [1]:
↦    _[1]=y2-y-1
↦    _[2]=x2+y+1
↦ [2]:
↦    _[1]=0
↦ [3]:
↦    _[1]=0
↦ [4]:
↦    _[1]=x
↦    _[2]=y
↦    _[3]=z
↦ [5]:
↦    _[1]=x
↦    _[2]=y
↦    _[3]=z
elimpart(i,3,1);
↦ [1]:
↦    _[1]=x4+3x2+1
↦ [2]:
↦    _[1]=y
↦ [3]:
↦    _[1]=x2+y+1
↦ [4]:
↦    _[1]=x
↦    _[2]=0
↦    _[3]=z
↦ [5]:
↦    _[1]=x
↦    _[2]=-x2-1
↦    _[3]=z
```

D.7.1.4 elimpartanyr

Procedure from library `presolve.lib` (see Section D.7.1 [presolve_lib], page 532).

Usage: elimpartanyr(i [,p,e]); i=ideal, p=polynomial, e=integer
p: product of vars to be eliminated,
e =0: substitute from linear part of i (same as elimlinearpart)
e!=0: eliminate also by direct substitution
(default: p=product of all vars, e=1)

Return: list of 6 objects:
[1]: (interreduced) ideal obtained by substituting from i those vars appearing in p, which occur in the linear part of i (or which can be expressed directly in the remaining variables, if e!=0)
[2]: ideal, variables which have been substituted
[3]: ideal, i-th element defines substitution of i-th var in [2]
[4]: ideal of variables of basering, substituted ones are set to 0

[5]: ideal, describing the map from the basering, say k[x(1..m)], to
itself onto k[..variables fom [4]..] and [1] is the image of i
[6]: int, # of vars considered for substitution (= # of factors of p)

The ideal i is generated by [1] and [3] in k[x(1..m)], the map [5] maps [3] to 0, hence induces an isomorphism

k[x(1..m)]/i -> k[..variables fom [4]..]/[1]

Note: the proc uses `execute` to create a ring with ordering dp and vars placed correctly and then applies `elimpart`.

Example:

```
LIB "presolve.lib";
ring s=0,(x,y,z),dp;
ideal i = x3+y2+z,x2y2+z3,y+z+1;
elimpartanyr(i,z);
↦ [1]:
↦    _[1]=x3+y2-y-1
↦    _[2]=x2y2-y3-3y2-3y-1
↦ [2]:
↦    _[1]=z
↦ [3]:
↦    _[1]=y+z+1
↦ [4]:
↦    _[1]=0
↦    _[2]=x
↦    _[3]=y
↦ [5]:
↦    _[1]=-y-1
↦    _[2]=x
↦    _[3]=y
↦ [6]:
↦    1
```

D.7.1.5 fastelim

Procedure from library `presolve.lib` (see Section D.7.1 [presolve_lib], page 532).

Usage: fastelim(i,p[h,o,a,b,e,m]); i=ideal, p=polynomial; h,o,a,b,e=integers p: product of variables to be eliminated;
Optional parameters:

- h !=0: use Hilbert-series driven std-basis computation
- o !=0: use proc `valvars` for a - hopefully - optimal ordering of vars
- a !=0: order vars to be eliminated w.r.t. increasing complexity
- b !=0: order vars not to be eliminated w.r.t. increasing complexity
- e !=0: use `elimpart` first to eliminate easy part
- m !=0: compute a minimal system of generators

(default: h,o,a,b,e,m = 0,1,0,0,0,0)

Return: ideal obtained from i by eliminating those variables, which occur in p

Example:

```
LIB "presolve.lib";
ring s=31991,(e,f,x,y,z,t,u,v,w,a,b,c,d),dp;
ideal i = w2+f2-1, x2+t2+a2-1,  y2+u2+b2-1, z2+v2+c2-1,
d2+e2-1, f4+2u, wa+tf, xy+tu+ab;
fastelim(i,xytua,1,1);       //with hilb,valvars
↦ _[1]=f2+w2-1
↦ _[2]=z2+v2+c2-1
↦ _[3]=e2+d2-1
fastelim(i,xytua,1,0,1);     //with hilb,minbase
↦ _[1]=z2+v2+c2-1
↦ _[2]=f2+w2-1
↦ _[3]=e2+d2-1
```

D.7.1.6 findvars

Procedure from library `presolve.lib` (see Section D.7.1 [presolve_lib], page 532).

Usage: findvars(id [,any]); id=poly/ideal/vector/module/matrix, any=any type

Return: if no second argument is present: ideal of variables occurring in id,
if a second argument is given (of any type): list L with 4 entries:

L[1]: ideal of variables occurring in id
L[2]: intvec of variables occurring in id
L[3]: ideal of variables not occurring in id
L[4]: intvec of variables not occurring in id

Example:

```
LIB "presolve.lib";
ring s  = 0,(e,f,x,y,t,u,v,w,a,d),dp;
ideal i = w2+f2-1, x2+t2+a2-1;
findvars(i);
↦ _[1]=f
↦ _[2]=x
↦ _[3]=t
↦ _[4]=w
↦ _[5]=a
findvars(i,1);
↦ [1]:
↦    _[1]=f
↦    _[2]=x
↦    _[3]=t
↦    _[4]=w
↦    _[5]=a
↦ [2]:
↦    2,3,5,8,9
↦ [3]:
↦    _[1]=e
↦    _[2]=y
↦    _[3]=u
↦    _[4]=v
↦    _[5]=d
↦ [4]:
↦    1,4,6,7,10
```

D.7.1.7 hilbvec

Procedure from library `presolve.lib` (see Section D.7.1 [presolve_lib], page 532).

Usage: hilbvec(id[,c,o]); id=poly/ideal/vector/module/matrix, c,o=strings,
c=char, o=ordering used by `hilb`
(default: c="32003", o="dp")

Return: intvec of 1-st Hilbert-series of id, computed in char c and ordering o

Note: id must be homogeneous (i.e. all vars have weight 1)

Example:

```
LIB "presolve.lib";
ring s   = 0,(e,f,x,y,z,t,u,v,w,a,b,c,d,H),dp;
ideal id = w2+f2-1, x2+t2+a2-1,  y2+u2+b2-1, z2+v2+c2-1,
d2+e2-1, f4+2u, wa+tf, xy+tu+ab;
id = homog(id,H);
hilbvec(id);
↦ 1,0,-7,0,20,0,-28,0,14,0,14,0,-28,0,20,0,-7,0,1,0
```

D.7.1.8 linearpart

Procedure from library `presolve.lib` (see Section D.7.1 [presolve_lib], page 532).

Usage: linearpart(id); id=ideal/module

Return: generators of id of total degree <= 1

Example:

```
LIB "presolve.lib";
ring r=0,(x,y,z),dp;
ideal i=1+x+x2+x3,3,x+3y+5z;
linearpart(i);
↦ _[1]=3
↦ _[2]=x+3y+5z
module m=[x,y,z],x*[x3,y2,z],[1,x2,z3,0,1];
show(linearpart(m));
↦ // module, 1 generator(s)
↦ [x,y,z]
```

D.7.1.9 tolessvars

Procedure from library `presolve.lib` (see Section D.7.1 [presolve_lib], page 532).

Usage: tolessvars(id [,s1,s2]); id poly/ideal/vector/module/matrix, s1,s2=strings
s1: name of new ring,
s2: new ordering
(default: s1="R(n)" where n is the # of vars in the new ring, s2="dp" or "ds" depending whether the first block of the old ordering is a p- resp. an s-ordering)

Create: nothing, if id contains all vars of the basering.
Else, create a ring with same char as the basering, but possibly less variables (only those variables which actually occur in id) and map id to the new ring, which will be the basering after the proc has finished.

Display: If printlevel >=0, display ideal of vars, which have been omitted from the old ring

Return: the original ideal id (see NOTE)

Note: You must not type, say, 'ideal id=tolessvars(id);' since the ring to which 'id' would belong will only be defined by the r.h.s.. But you may type 'def id=tolessvars(id);' or 'list id=tolessvars(id);' since then 'id' does not a priory belong to a ring, its type will be defined by the right hand side. Moreover, do not use a name which occurs in the old ring, for the same reason.

Example:

```
LIB "presolve.lib";
ring r  = 0,(x,y,z),dp;
ideal i = y2-x3,x-3,y-2x;
def j   = tolessvars(i,"R_r","lp");
↦
↦ // variables which did not occur:
↦ z
↦ // basering is now R_r
show(basering);
↦ // ring: (0),(x,y),(lp(2),C);
↦ // minpoly = 0
↦ // objects belonging to this ring:
↦ // j                    [0]  ideal, 3 generator(s)
j;
↦ j[1]=-x3+y2
↦ j[2]=x-3
↦ j[3]=-2x+y
kill R_r;
```

D.7.1.10 solvelinearpart

Procedure from library `presolve.lib` (see Section D.7.1 [presolve_lib], page 532).

Usage: solvelinearpart(id [,n]); id=ideal/module, n=integer,
(default: n=0)

Return: (interreduced) generators of id of degree <=1 in reduced triangular form if n=0 [non-reduced triangular form if n!=0]

Assume: monomial ordering is a global ordering (p-ordering)

Note: may be used to solve a system of linear equations see proc `gauss_row` from 'matrix.lib' for a different method

Warning: the result is very likely to be false for 'real' coefficients, use char 0 instead!

Example:

```
LIB "presolve.lib";
// Solve the system of linear equations:
//         3x +   y +  z -  u = 2
//         3x +  8y + 6z - 7u = 1
//        14x + 10y + 6z - 7u = 0
//         7x +  4y + 3z - 3u = 3
ring r = 0,(x,y,z,u),lp;
ideal i= 3x +   y +  z -  u,
13x +  8y + 6z - 7u,
14x + 10y + 6z - 7u,
```

```
7x +  4y + 3z - 3u;
ideal j= 2,1,0,3;
j = i-j;                         // difference of 1x4 matrices
// compute reduced triangular form, setting
solvelinearpart(j);              // the RHS equal 0 gives the solutions!
↦ _[1]=u-4
↦ _[2]=z-4
↦ _[3]=y+1
↦ _[4]=x-1
solvelinearpart(j,1); "";        // triangular form, not reduced
↦ _[1]=u-4
↦ _[2]=3z-8u+20
↦ _[3]=18y-6z+7u+14
↦ _[4]=13x+8y+6z-7u-1
↦
```

D.7.1.11 sortandmap

Procedure from library `presolve.lib` (see Section D.7.1 [presolve_lib], page 532).

Usage: sortandmap(id,s1,s2[,n1,p1,n2,p2...,o1,m1,o2,m2...]);
id=poly/ideal/vector/module,
s1,s2 = strings (names for new ring and mapped id),
p1,p2,...= polynomials (product of variables),
n1,n2,...= integers,
o1,o2,...= strings,
m1,m2,...= integers
(default: p1=product of all vars, n1=0, o1="dp",m1=0)
the last pi (containing the remaining vars) may be omitted

Create: a new ring and map id into it, the new ring has same char as basering but with new ordering and vars sorted in the following manner:
- each block of vars occurring in pi is sorted w.r.t. its complexity in id,
- ni controls the sorting in i-th block (= vars occurring in pi):
  ni=0 (resp.!=0) means that less (resp. more) complex vars come first
- oi and mi define the monomial ordering of the i-th block:
  if mi =0, oi=ordstr(i-th block)
  if mi!=0, the ordering of the i-th block itself is a blockordering,
    each subblock having ordstr=oi, such that vars of same complexity are
    in one block

Note that only simple ordstrings oi are allowed:
"lp","dp","Dp","ls","ds","Ds".

Return: nothing

Note: We define a variable x to be more complex than y (with respect to id) if val(x) > val(y) lexicographically, where val(x) denotes the valuation vector of x:
consider id as list of polynomials in x with coefficients in the remaining variables. Then:
val(x) = (maximal occurring power of x, # of all monomials in leading coefficient, # of all monomials in coefficient of next smaller power of x,...).

Example:

```
LIB "presolve.lib";
```

```
ring s = 32003,(x,y,z),dp;
ideal i=x3+y2,xz+z2;
sortandmap(i,"R_r","i");
// i is now an ideal in the new basering R_r
show(R_r);
↦ // ring: (32003),(y,z,x),(dp(3),C);
↦ // minpoly = 0
↦ // objects belonging to this ring:
↦ // i                    [0]  ideal, 2 generator(s)
kill R_r; setring s;
sortandmap(i,"R_r","i",1,xy,0,z,0,"ds",0,"lp",0);
show(R_r);
↦ // ring: (32003),(x,y,z),(ds(2),lp(1),C);
↦ // minpoly = 0
↦ // objects belonging to this ring:
↦ // i                    [0]  ideal, 2 generator(s)
kill R_r;
```

D.7.1.12 sortvars

Procedure from library `presolve.lib` (see Section D.7.1 [presolve_lib], page 532).

Usage: sortvars(id[,n1,p1,n2,p2,...]);
id=poly/ideal/vector/module,
p1,p2,...= polynomials (product of vars),
n1,n2,...=integers
(default: p1=product of all vars, n1=0)
the last pi (containing the remaining vars) may be omitted

Compute: sort variables with respect to their complexity in id

Return: list of two elements, an ideal and a list:
[1]: ideal, variables of basering sorted w.r.t their complexity in id
ni controls the ordering in i-th block (= vars occurring in pi):
ni=0 (resp.!=0) means that less (resp. more) complex vars come first
[2]: a list with 4 entries for each pi:
ideal ai : vars of pi in correct order,
intvec vi: permutation vector describing the ordering in ai,
intmat Mi: valuation matrix of ai, the columns of Mi being the
valuation vectors of the vars in ai
intvec wi: size of 1-st, 2-nd,... block of identical columns of Mi
(vars with same valuation)

Note: We define a variable x to be more complex than y (with respect to id) if val(x) > val(y) lexicographically, where val(x) denotes the valuation vector of x:
consider id as list of polynomials in x with coefficients in the remaining variables. Then:
val(x) = (maximal occurring power of x, # of all monomials in leading coefficient, # of all monomials in coefficient of next smaller power of x,...).

Example:

```
LIB "presolve.lib";
ring s=0,(x,y,z,w),dp;
ideal i = x3+y2+yw2,xz+z2,xyz-w2;
sortvars(i,0,xy,1,zw);
```

```
↦ [1]:
↦    _[1]=y
↦    _[2]=x
↦    _[3]=w
↦    _[4]=z
↦ [2]:
↦    [1]:
↦       _[1]=y
↦       _[2]=x
↦    [2]:
↦       2,1
↦    [3]:
↦       2,3,
↦       1,1,
↦       2,0,
↦       0,2
↦    [4]:
↦       1,1
↦    [5]:
↦       _[1]=w
↦       _[2]=z
↦    [6]:
↦       2,1
↦    [7]:
↦       2,2,
↦       2,1,
↦       0,2
↦    [8]:
↦       1,1
```

D.7.1.13 shortid

Procedure from library `presolve.lib` (see Section D.7.1 [presolve_lib], page 532).

Usage: shortid(id,n[,e]); id= ideal/module, n,e=integers

Return: - if called with two arguments or e=0:
same type as id, containing generators of id having <= n terms.
- if called with three arguments and e!=0:
a list L:
L[1]: same type as id, containing generators of id having <= n terms.
L[2]: number of corresponding generator of id

Note: May be used to compute partial standard basis in case id is to hard

Example:

```
LIB "presolve.lib";
ring s=0,(x,y,z,w),dp;
ideal i = (x3+y2+yw2)^2,(xz+z2)^2,xyz-w2-xzw;
shortid(i,3);
↦ _[1]=x2z2+2xz3+z4
↦ _[2]=xyz-xzw-w2
```

D.7.1.14 valvars

Procedure from library `presolve.lib` (see Section D.7.1 [presolve_lib], page 532).

Usage: valvars(id[,n1,p1,n2,p2,...]);
id=poly/ideal/vector/module,
p1,p2,...= polynomials (product of vars),
n1,n2,...= integers,

ni controls the ordering of vars occurring in pi: ni=0 (resp.!=0) means that less (resp. more) complex vars come first
(default: p1=product of all vars, n1=0)
the last pi (containing the remaining vars) may be omitted

Compute: valuation (complexity) of variables with respect to id.
ni controls the ordering of vars occurring in pi:
ni=0 (resp.!=0) means that less (resp. more) complex vars come first.

Return: list with 3 entries:
[1]: intvec, say v, describing the permutation such that the permuted
  ringvariables are ordered with respect to their complexity in id
[2]: list of intvecs, i-th intvec, say v(i) describing permutation
  of vars in a(i) such that v=v(1),v(2),...
[3]: list of ideals and intmat's, say a(i) and M(i), where
  a(i): factors of pi,
  M(i): valuation matrix of a(i), such that the j-th column of M(i)
    is the valuation vector of j-th generator of a(i)

Note: Use `sortvars` in order to actually sort the variables! We define a variable x to be more complex than y (with respect to id) if val(x) > val(y) lexicographically, where val(x) denotes the valuation vector of x:
consider id as list of polynomials in x with coefficients in the remaining variables. Then:
val(x) = (maximal occurring power of x, # of all monomials in leading coefficient, # of all monomials in coefficient of next smaller power of x,...).

Example:

```
LIB "presolve.lib";
ring s=0,(x,y,z,a,b),dp;
ideal i=ax2+ay3-b2x,abz+by2;
valvars (i,0,xyz);
↦ [1]:
↦    3,1,2,4,5
↦ [2]:
↦    [1]:
↦       3,1,2
↦    [2]:
↦       1,2
↦ [3]:
↦    [1]:
↦       _[1]=x
↦       _[2]=y
↦       _[3]=z
↦    [2]:
↦       2,3,1,
↦       1,1,1,
↦       1,1,0
↦    [3]:
↦       _[1]=a
```

```
↦       _[2]=b
↦    [4]:
↦       1,2,
↦       3,1,
↦       0,2
```

D.7.1.15 idealSimplify

Procedure from library `presolve.lib` (see Section D.7.1 [presolve_lib], page 532).

Usage: idealSimplify(id); id ideal

Return: ideal I = eliminate(Id,m) m is a product of variables which are only linearly involved in the generators of id

Example:

```
LIB "presolve.lib";
ring r=0,(x,y,z,w,t),dp;
ideal i=
t,
x3+y2+2z,
x2+3y,
x2+y2+z2,
w2+z;
ideal j=idealSimplify(i);
ideal k=eliminate(i,zyt);
reduce(k,std(j));
↦ _[1]=0
↦ _[2]=0
reduce(j,std(k));
↦ _[1]=0
↦ _[2]=0
```

D.7.1.16 idealSplit

Procedure from library `presolve.lib` (see Section D.7.1 [presolve_lib], page 532).

Usage: idealSplit(id,timeF,timeS); id ideal and optional
timeF ,timeS integers to bound the time which can be used for factorization resp. standard basis computation

Return: a list of ideals such that their intersection
has the same radical as id

Example:

```
LIB "presolve.lib";
ring r=32003,(b,s,t,u,v,w,x,y,z),dp;
ideal i=
bv+su,
bw+tu,
sw+tv,
by+sx,
bz+tx,
sz+ty,
uy+vx,
```

```
uz+wx,
vz+wy,
bvz;
idealSplit(i);
↦ [1]:
↦    _[1]=x
↦    _[2]=u
↦    _[3]=t
↦    _[4]=s
↦    _[5]=b
↦    _[6]=wy+vz
↦ [2]:
↦    _[1]=z
↦    _[2]=w
↦    _[3]=t
↦    _[4]=s
↦    _[5]=b
↦    _[6]=vx+uy
↦ [3]:
↦    _[1]=z
↦    _[2]=x
↦    _[3]=w
↦    _[4]=u
↦    _[5]=t
↦    _[6]=b
↦ [4]:
↦    _[1]=z
↦    _[2]=y
↦    _[3]=x
↦    _[4]=t
↦    _[5]=s
↦    _[6]=b
↦ [5]:
↦    _[1]=z
↦    _[2]=y
↦    _[3]=x
↦    _[4]=u
↦    _[5]=b
↦    _[6]=tv+sw
↦ [6]:
↦    _[1]=z
↦    _[2]=y
↦    _[3]=x
↦    _[4]=w
↦    _[5]=t
↦    _[6]=su+bv
↦ [7]:
↦    _[1]=w
↦    _[2]=v
↦    _[3]=u
↦    _[4]=t
↦    _[5]=s
↦    _[6]=b
```

```
↦ [8]:
↦    _[1]=x
↦    _[2]=w
↦    _[3]=v
↦    _[4]=u
↦    _[5]=b
↦    _[6]=ty+sz
↦ [9]:
↦    _[1]=z
↦    _[2]=w
↦    _[3]=v
↦    _[4]=u
↦    _[5]=t
↦    _[6]=sx+by
↦ [10]:
↦    _[1]=z
↦    _[2]=y
↦    _[3]=x
↦    _[4]=w
↦    _[5]=v
↦    _[6]=u
↦ [11]:
↦    _[1]=y
↦    _[2]=v
↦    _[3]=t
↦    _[4]=s
↦    _[5]=b
↦    _[6]=wx+uz
↦ [12]:
↦    _[1]=y
↦    _[2]=x
↦    _[3]=v
↦    _[4]=u
↦    _[5]=s
↦    _[6]=b
↦ [13]:
↦    _[1]=z
↦    _[2]=y
↦    _[3]=x
↦    _[4]=v
↦    _[5]=s
↦    _[6]=tu+bw
↦ [14]:
↦    _[1]=z
↦    _[2]=y
↦    _[3]=w
↦    _[4]=v
↦    _[5]=t
↦    _[6]=s
↦ [15]:
↦    _[1]=y
↦    _[2]=w
↦    _[3]=v
```

```
↦    _[4]=u
↦    _[5]=s
↦    _[6]=tx+bz
```

D.7.2 solve_lib

Library: solve.lib

Purpose: Complex Solving of Polynomial Systems

Author: Moritz Wenk, email: wenk@mathematik.uni-kl.de
Wilfred Pohl, email: pohl@mathematik.uni-kl.de

Procedures:

D.7.2.1 laguerre_solve

Procedure from library `solve.lib` (see Section D.7.2 [solve_lib], page 546).

Usage: laguerre_solve(f [, m, l, n, s]); f = polynomial,
m, l, n, s = integers (control parameters of the method)

m: precision of output in digits (4 <= m), if basering is not
the ring of complex numbers;
(if s != 0, m defines also the precision of the residual error
i.e. | f(root) | should be <= 0.1^m).
l: precision of internal computation in decimal digits (l >=8)
only if the basering is not complex or complex with smaller
precision,
n: control of multiplicity of roots or of splitting of f into
squarefree factors
n < 0, no split of f (good, if all roots are simple)
n >= 0, try to split
n = 0, return only different roots
n > 0, find all roots (with multiplicity)
s: s != 0, returns ERROR if | f(root) | > 0.1^m
(default: m, l, n, s = 8, 30, 1, 0)

Assume: f is an univariate polynomial

Return: list of (complex) roots of the polynomial f, depending on n.

The result is
of type string: if the basering is not complex,
of type number: otherwise.

Note: If printlevel >0: displays comments (default = 0).
If s != 0 and if the procedure stops with ERROR, try a higher internal precision m.

Example:

```
LIB "solve.lib";
// Find all roots of an univariate polynomial using Laguerre's method:
ring rs1= 0,(x,y),lp;
poly f = 15x5 + x3 + x2 - 10;
// 10 digits precision
laguerre_solve(f,10);
```

```
↦ [1]:
↦    (0.2930464644-i*0.9003002396)
↦ [2]:
↦    (0.2930464644+i*0.9003002396)
↦ [3]:
↦    (-0.7392783383-i*0.5355190078)
↦ [4]:
↦    (-0.7392783383+i*0.5355190078)
↦ [5]:
↦    0.8924637479
// Now with complex coefficients,
// internal precision is 30 digits (default)
printlevel=2;
ring rsc= (real,10,i),x,lp;
poly f = (15.4+i*5)*x^5 + (25.0e-2+i*2)*x^3 + x2 - 10*i;
list l = laguerre_solve(f);
↦ //BEGIN laguerre_solve
↦ //control: complex ring with precision 30
↦ //working in:  ring lagc=(complex,30,30),x,lp;
↦ //         polynomial has complex coefficients
↦ //split in working ring:
↦ // split without result
↦ //END laguerre_solve
l;
↦ [1]:
↦    (0.04588498039+i*0.9133296179)
↦ [2]:
↦    (0.5037408279-i*0.8058051828)
↦ [3]:
↦    (-0.5462895588-i*0.6796668873)
↦ [4]:
↦    (0.8524014357+i*0.2163760334)
↦ [5]:
↦    (-0.8557376852+i*0.3557664188)
// check result, value of substituted poly should be near to zero
// remember that l contains a list of strings
// in the case of a different ring
subst(f,x,l[1]);
↦ 0
subst(f,x,l[2]);
↦ 0
```

D.7.2.2 solve

Procedure from library `solve.lib` (see Section D.7.2 [solve_lib], page 546).

Usage: solve(G [, m, n, l]); G = ideal,
m, n, l = integers (control parameters of the method)

m: precision of output in digits (4 <= m) and of the
generated ring of complex numbers;

n: control of multiplicity
n = 0, return all different roots
n != 0, find all roots (with multiplicity)

l: precision of internal computation in decimal digits (l >=8)
only if the basering is not complex or complex with smaller
precision,
(default: m, n, l = 8, 0, 30 or
for (n != 0 and size(#) = 2) l = 60)

Assume: the ground field has char 0;
the ideal is 0-dimensional

Return: list of solutions of the ideal G, depending on n,
one solution is a list of complex numbers in the generated
output ring (the new basering).

The result is a list L
n = 0: a list of all different solutions (L[i]),
n != 0: a list of two elements,
L[i][1] contains all different solution with
the same multiplicity
L[i][2] the multiplicity
L is ordered w.r.t. multiplicity (the smallest first).

Note: If the problem is not 0-dim. he procedure stops with ERROR,
if the ideal G is not a lex. standard basis, it is
generated with internal computation (Hilbert driven),
if the input-ring (with char 0) has the name "<A>", the
lex. and complex output-ring has the name "<A>C".

Example:

```
LIB "solve.lib";
// Find all roots of a multivariate ideal using triangular sets:
int d=4;// with these 3 parameters you may construct
int t=3;// very hard problems for 'solve'
int s=2;
int i;
ring A=0,(x(1..d)),dp;
poly p=-1;
for(i=d;i>0;i--){p=p+x(i)^s;}
ideal I=x(d)^t-x(d)^s+p;
for(i=d-1;i>0;i--){I=x(i)^t-x(i)^s+p,I;}
I;
↦ I[1]=x(1)^3+x(2)^2+x(3)^2+x(4)^2-1
↦ I[2]=x(2)^3+x(1)^2+x(3)^2+x(4)^2-1
↦ I[3]=x(3)^3+x(1)^2+x(2)^2+x(4)^2-1
↦ I[4]=x(4)^3+x(1)^2+x(2)^2+x(3)^2-1
// the mutiplicity is
vdim(std(I));
↦ 81
list l1=solve(I,6,0);
↦ // name of new current ring: AC
// the current ring is
AC;
↦ //   characteristic : 0 (complex:6 digits, additional 6 digits)
↦ //   1 parameter    : i
```

```
↦ //   minpoly        : (i^2+1)
↦ //   number of vars : 4
↦ //        block   1 : ordering lp
↦ //                  : names    x(1) x(2) x(3) x(4)
↦ //        block   2 : ordering C
// you must start with char. 0
setring A;
list l2=solve(I,6,1);
↦ // name of current ring: AC
// the number of different solutions is
size(l1);
↦ 37
// this is equal to
size(l2[1][1])+size(l2[2][1]);
↦ 37
// the number of solutions with multiplicity is
size(l2[1][1])*l2[1][2]+size(l2[2][1])*l2[2][2];
↦ 81
// the solutions with multiplicity
l2[2][2];
↦ 12
// are
l2[2][1];
↦ [1]:
↦    [1]:
↦       0
↦    [2]:
↦       0
↦    [3]:
↦       1
↦    [4]:
↦       0
↦ [2]:
↦    [1]:
↦       0
↦    [2]:
↦       1
↦    [3]:
↦       0
↦    [4]:
↦       0
↦ [3]:
↦    [1]:
↦       1
↦    [2]:
↦       0
↦    [3]:
↦       0
↦    [4]:
↦       0
↦ [4]:
↦    [1]:
↦       0
```

```
↦     [2]:
↦        0
↦     [3]:
↦        0
↦     [4]:
↦        1
```

D.7.2.3 ures_solve

Procedure from library `solve.lib` (see Section D.7.2 [solve_lib], page 546).

Usage: ures_solve(i [, k, p]); i = ideal, k, p = integers
k=0: use sparse resultant matrix of Gelfand, Kapranov and Zelevinsky,
k=1: use resultant matrix of Macaulay which works only for
homogeneous ideals,
p>0: defines precision of the long floats for internal computation
if the basering is not complex (in decimal digits),
(default: k, p = 0, 30)

Assume: i is a zerodimensional ideal with
nvars(basering) = ncols(i) = number of vars
actually occurring in i,

Return: list of all (complex) roots of the polynomial system i = 0,
the result is
of type string: if the basering is not complex,
of type number: otherwise.

Example:

```
LIB "solve.lib";
// compute the intersection points of two curves
ring rsq = 0,(x,y),lp;
ideal gls=  x2 + y2 - 10, x2 + xy + 2y2 - 16;
ures_solve(gls,0,16);
↦ [1]:
↦     [1]:
↦        1
↦     [2]:
↦        -3
↦ [2]:
↦     [1]:
↦        -1
↦     [2]:
↦        3
↦ [3]:
↦     [1]:
↦        2.82842712474619
↦     [2]:
↦        1.414213562373095
↦ [4]:
↦     [1]:
↦        -2.82842712474619
```

```
↦    [2]:
↦       -1.414213562373095
// result is a list (x,y)-coordinates as strings
// now with complex coefficient field, precision is 20 digits
ring rsc= (real,20,I),(x,y),lp;
ideal i = (2+3*I)*x2 + (0.35+I*45.0e-2)*y2 - 8, x2 + xy + (42.7)*y2;
list l= ures_solve(i,0,10);
// result is a list of (x,y)-coordinates of complex numbers
l;
↦ [1]:
↦    [1]:
↦       (-1.315392899374542198+I*0.70468233142752928117)
↦    [2]:
↦       (0.12292646536251281054+I*0.19245727404407015049)
↦ [2]:
↦    [1]:
↦       (1.315392899374542198-I*0.70468233142752928117)
↦    [2]:
↦       (-0.12292646536251281054-I*0.19245727404407015049)
↦ [3]:
↦    [1]:
↦       (1.31584587549391830705-I*0.70396753310002259573)
↦    [2]:
↦       (0.092006639590217681983+I*0.20902112035965287775)
↦ [4]:
↦    [1]:
↦       (-1.31584587549391830705+I*0.70396753310002259573)
↦    [2]:
↦       (-0.092006639590217681983-I*0.20902112035965287775)
// check the result
subst(subst(i[1],x,l[1][1]),y,l[1][2]);
↦ 0
```

D.7.2.4 mp_res_mat

Procedure from library `solve.lib` (see Section D.7.2 [solve_lib], page 546).

Usage: mp_res_mat(i [, k]); i ideal, k integer,

k=0: sparse resultant matrix of Gelfand, Kapranov and Zelevinsky,
k=1: resultant matrix of Macaulay (k=0 is default)

Assume:

the number of elements in the input system must be
the number of variables in the basering plus one,
if k=1 then i must be homogeneous,

Return: module representing the multipolynomial resultant matrix

Example:

```
LIB "solve.lib";
// compute resultant matrix in ring with parameters (sparse resultant matrix)
ring rsq= (0,u0,u1,u2),(x1,x2),lp;
```

```
ideal i= u0+u1*x1+u2*x2,x1^2 + x2^2 - 10,x1^2 + x1*x2 + 2*x2^2 - 16;
module m = mp_res_mat(i);
print(m);
↦ -16,0,  -10,0,  (u0),0,   0,  0,   0,   0,
↦ 0,  -16,0,  -10,(u2),(u0),0,  0,   0,   0,
↦ 2,  0,  1,  0,  0,   (u2),0,  0,   0,   0,
↦ 0,  2,  0,  1,  0,   0,   0,  0,   0,   0,
↦ 0,  0,  0,  0,  (u1),0,   -10,(u0),0,   -16,
↦ 1,  0,  0,  0,  0,   (u1),0,  (u2),(u0),0,
↦ 0,  1,  0,  0,  0,   0,   1,  0,   (u2),2,
↦ 1,  0,  1,  0,  0,   0,   0,  (u1),0,   0,
↦ 0,  1,  0,  1,  0,   0,   0,  0,   (u1),1,
↦ 0,  0,  0,  0,  0,   0,   1,  0,   0,   1
// computing sparse resultant
det(m);
↦ (-2*u0^4+18*u0^2*u1^2+4*u0^2*u1*u2+22*u0^2*u2^2-16*u1^4+80*u1^3*u2-52*u1^\
   2*u2^2-120*u1*u2^3-36*u2^4)
// compute resultant matrix (Macaulay resultant matrix)
ring rdq= (0,u0,u1,u2),(x0,x1,x2),lp;
ideal h=  homog(imap(rsq,i),x0);
h;
↦ h[1]=(u0)*x0+(u1)*x1+(u2)*x2
↦ h[2]=-10*x0^2+x1^2+x2^2
↦ h[3]=-16*x0^2+x1^2+x1*x2+2*x2^2
module m = mp_res_mat(h,1);
print(m);
↦ x0, x1, x2, 0, 0, 0, 0,0, 0, 0,
↦ 0,  x0, 0,  x1,x2,0, 0,0, 0, 0,
↦ 0,  0,  x0, 0, x1,x2,0,0, 0, 0,
↦ -10,0,  0,  1, 0, 1, 0,0, 0, 0,
↦ 0,  0,  0,  0, x0,0, 0,x1,x2,0,
↦ -16,0,  0,  1, 1, 2, 0,0, 0, 0,
↦ 0,  -10,0,  0, 0, 0, 1,0, 1, 0,
↦ 0,  0,  -10,0, 0, 0, 0,1, 0, 1,
↦ 0,  -16,0,  0, 0, 0, 1,1, 2, 0,
↦ 0,  0,  -16,0, 0, 0, 0,1, 1, 2
// computing Macaulay resultant (should be the same as above!)
det(m);
↦ 2*x0^4-18*x0^2*x1^2-4*x0^2*x1*x2-22*x0^2*x2^2+16*x1^4-80*x1^3*x2+52*x1^2*\
   x2^2+120*x1*x2^3+36*x2^4
// compute numerical sparse resultant matrix
setring rsq;
ideal ir= 15+2*x1+5*x2,x1^2 + x2^2 - 10,x1^2 + x1*x2 + 2*x2^2 - 16;
module mn = mp_res_mat(ir);
print(mn);
↦ 15,0, -10,0,  0, 0, 0,  -16,0,  0,
↦ 5, 15,0,  -10,0, 0, 0,  0,  -16,0,
↦ 0, 5, 1,  0,  0, 0, 0,  2,  0,  0,
↦ 0, 0, 0,  1,  0, 0, 0,  0,  2,  0,
↦ 2, 0, 0,  0,  15,0, -10,0,  0,  -16,
↦ 0, 2, 0,  0,  5, 15,0,  1,  0,  0,
↦ 0, 0, 0,  0,  0, 5, 1,  0,  1,  2,
↦ 0, 0, 1,  0,  2, 0, 0,  1,  0,  0,
```

```
↦ 0, 0, 0,  1,  0, 2, 0,  0,  1,  1,
↦ 0, 0, 0,  0,  0, 0, 1,  0,  0,  1
// computing sparse resultant
det(mn);
↦ -7056
```

D.7.2.5 interpolate

Procedure from library `solve.lib` (see Section D.7.2 [solve_lib], page 546).

Usage: interpolate(p,v,d); p,v=ideals of numbers, d=integer

Assume:

ground field K are the rational numbers,
p and v consists of elements of the ground field K
with the following number of elements:
size(p) = n and size(v)=N=(d+1)^n with n equals
the number of variables,
p is considered as point in K^n and with
pi=(p[1]^i,..,p[n]^i), i=1,..,N the returned
polynomial f satisfies f(pi) = v[i]
(p[j] != -1,0,1);

Return:

the unique polynomial f of degree n*d with prescribed
values v at certain points p1,..,pN derived from p;

Note: mainly useful when n=1, i.e. f is satisfying

f(p^(i-1)) = v[i], i=1..d+1

Example:

```
LIB "solve.lib";
ring r1 = 0,(x),lp;
// determine f with deg(f) = 4 and
// v = values of f at points 3^0, 3^1, 3^2, 3^3, 3^4
ideal v=16,0,11376,1046880,85949136;
interpolate( 3, v, 4 );
↦ 2x4-22x2+36
```

D.7.2.6 fglm_solve

Procedure from library `solve.lib` (see Section D.7.2 [solve_lib], page 546).

Usage: fglm_solve(i [, p]); i=ideal, p=integer,

p>0: gives precision of complex numbers
in decimal digits (default: p=30),

Assume: the ground field has char 0.

Return:

a list of complex roots of type number, the procedure
uses a standard basis of i to determine all complex
roots of i.

Note:

The procedure creates a ring rC with the same number
of variables but with complex coefficients (and precision p).

Example:

```
LIB "solve.lib";
ring r = 0,(x,y),lp;
// compute the intersection points of two curves
ideal s=  x2 + y2 - 10, x2 + xy + 2y2 - 16;
fglm_solve(s,10);
↦ // name of new ring: rC
↦ // list of roots: rlist
rlist;
↦ [1]:
↦    [1]:
↦       2.8284271247
↦    [2]:
↦       1.4142135624
↦ [2]:
↦    [1]:
↦       -2.8284271247
↦    [2]:
↦       -1.4142135624
↦ [3]:
↦    [1]:
↦       1
↦    [2]:
↦       -3
↦ [4]:
↦    [1]:
↦       -1
↦    [2]:
↦       3
```

D.7.2.7 lex_solve

Procedure from library `solve.lib` (see Section D.7.2 [solve_lib], page 546).

Usage: lex_solve(i[,p]); i=ideal, p=integer,

p>0: gives precision of complex numbers
in decimal digits (default: p=30),

Assume:

i is a reduced lexicographical Groebner bases of a
zero-dimensional ideal (i), sorted by increasing leading terms.

Return: nothing

Create:

The procedure creates a complec ring with the same variables but
with complex coefficients (and precision p).

In this ring a list rlist of numbers is created,
in which the complex roots of i are stored.

Example:

```
LIB "solve.lib";
ring r = 0,(x,y),lp;
// compute the intersection points of two curves
ideal s=  x2 + y2 - 10, x2 + xy + 2y2 - 16;
lex_solve(stdfglm(s),10);
↦ // name of new ring: rC
↦ // list of roots: rlist
rlist;
↦ [1]:
↦    [1]:
↦       2.8284271247
↦    [2]:
↦       1.4142135624
↦ [2]:
↦    [1]:
↦       -2.8284271247
↦    [2]:
↦       -1.4142135624
↦ [3]:
↦    [1]:
↦       1
↦    [2]:
↦       -3
↦ [4]:
↦    [1]:
↦       -1
↦    [2]:
↦       3
```

D.7.2.8 triangLf_solve

Procedure from library `solve.lib` (see Section D.7.2 [solve_lib], page 546).

Usage: triangLf_solve(i [, p]); i ideal, p integer,
p>0: gives precision of complex numbers in digits,
(default: p=30)

Assume: the ground field has char 0;
i zero-dimensional ideal

Return: nothing

Create: The procedure creates a ring rC with the same number of variables but with complex coefficients (and precision p).
In rC a list rlist of numbers is created, in which the complex roots of i are stored.
The proc uses a triangular system (Lazard's Algorithm with factorization) computed from a standard basis to determine recursively all complex roots with Laguerre's algorithm of input ideal i.

Example:

```
LIB "solve.lib";
ring r = 0,(x,y),lp;
// compute the intersection points of two curves
ideal s=  x2 + y2 - 10, x2 + xy + 2y2 - 16;
triangLf_solve(s,10);
↦ // name of new ring: rC
↦ // list of roots: rlist
rlist;
↦ [1]:
↦    [1]:
↦       -1
↦    [2]:
↦       3
↦ [2]:
↦    [1]:
↦       1
↦    [2]:
↦       -3
↦ [3]:
↦    [1]:
↦       2.8284271247
↦    [2]:
↦       1.4142135624
↦ [4]:
↦    [1]:
↦       -2.8284271247
↦    [2]:
↦       -1.4142135624
```

D.7.2.9 triangM_solve

Procedure from library `solve.lib` (see Section D.7.2 [solve_lib], page 546).

Usage: triangM_solve(i [, p]); i=ideal, p=integer,
p>0: gives precision of complex numbers in digits,
(default: p=30)

Assume: the ground field has char 0;
i zero-dimensional ideal

Return: nothing

Create: The procedure creates a ring rC with the same number of variables but with complex coefficients (and precision p).
In rC a list rlist of numbers is created, in which the complex roots of i are stored.
The proc uses a triangular system (Moellers Algorithm) computed from a standard basis to determine recursively all complex roots with Laguerre's algorithm of input ideal i.

Example:

```
LIB "solve.lib";
ring r = 0,(x,y),lp;
// compute the intersection points of two curves
ideal s=  x2 + y2 - 10, x2 + xy + 2y2 - 16;
triangM_solve(s,10);
```

```
↦ // name of new ring: rC
↦ // list of roots: rlist
rlist;
↦ [1]:
↦    [1]:
↦       2.8284271247
↦    [2]:
↦       1.4142135624
↦ [2]:
↦    [1]:
↦       -2.8284271247
↦    [2]:
↦       -1.4142135624
↦ [3]:
↦    [1]:
↦       1
↦    [2]:
↦       -3
↦ [4]:
↦    [1]:
↦       -1
↦    [2]:
↦       3
```

D.7.2.10 triangL_solve

Procedure from library `solve.lib` (see Section D.7.2 [solve_lib], page 546).

Usage: triangL_solve(i [, p]); i=ideal, p=integer,
p>0: gives precision of complex numbers in digits,
(default: p=30)

Assume: the ground field has char 0;
i zero-dimensional ideal

Return: nothing

Create: The procedure creates a ring rC with the same number of variables but with complex coefficients (and precision p).
In rC a list rlist of numbers is created, in which the complex roots of i are stored.
The proc uses a triangular system (Lazard's Algorithm) computed from a standard basis to determine recursively all complex roots with Laguerre's algorithm of input ideal i.

Example:

```
LIB "solve.lib";
ring r = 0,(x,y),lp;
// compute the intersection points of two curves
ideal s=  x2 + y2 - 10, x2 + xy + 2y2 - 16;
triangL_solve(s,10);
↦ // name of new ring: rC
↦ // list of roots: rlist
rlist;
↦ [1]:
↦    [1]:
```

```
↦       2.8284271247
↦    [2]:
↦       1.4142135624
↦ [2]:
↦    [1]:
↦       -2.8284271247
↦    [2]:
↦       -1.4142135624
↦ [3]:
↦    [1]:
↦       1
↦    [2]:
↦       -3
↦ [4]:
↦    [1]:
↦       -1
↦    [2]:
↦       3
```

D.7.2.11 triang_solve

Procedure from library `solve.lib` (see Section D.7.2 [solve_lib], page 546).

Usage: triang_solve(l,p [, d]); l=list, p,d=integers,
l a list of finitely many triangular systems, such that the union of their varieties equals the variety of the initial ideal.
p>0: gives precision of complex numbers in digits,
d>0: gives precision (1<d<p) for near-zero-determination,
(default: d=1/2*p)

Assume: the ground field has char 0;
l was computed using Algorithm of Lazard or Algorithm of Moeller (see triang.lib).

Return: nothing

Create: The procedure creates a ring rC with the same number of variables but with complex coefficients (and precision p).
In rC a list rlist of numbers is created, in which the complex roots of i are stored.

Example:

```
LIB "solve.lib";
ring r = 0,(x,y),lp;
// compute the intersection points of two curves
ideal s=  x2 + y2 - 10, x2 + xy + 2y2 - 16;
triang_solve(triangLfak(stdfglm(s)),10);
↦ // name of new ring: rC
↦ // list of roots: rlist
rlist;
↦ [1]:
↦    [1]:
↦       -1
↦    [2]:
↦       3
↦ [2]:
```

```
↦     [1]:
↦        1
↦     [2]:
↦        -3
↦  [3]:
↦     [1]:
↦        2.8284271247
↦     [2]:
↦        1.4142135624
↦  [4]:
↦     [1]:
↦        -2.8284271247
↦     [2]:
↦        -1.4142135624
```

D.7.3 triang_lib

Library: triang.lib

Purpose: Decompose Zero-dimensional Ideals into Triangular Sets

Author: D. Hillebrand

Procedures:

D.7.3.1 triangL

Procedure from library `triang.lib` (see Section D.7.3 [triang_lib], page 559).

Usage: triangL(G); G=ideal

Assume: G is the reduced lexicographical Groebner bases of the zero-dimensional ideal (G), sorted by increasing leading terms.

Return: a list of finitely many triangular systems, such that the union of their varieties equals the variety of (G).

Note: Algorithm of Lazard (see: Lazard, D.: Solving zero-dimensional algebraic systems, J. Symb. Comp. 13, 117 - 132, 1992).

Example:

```
LIB "triang.lib";
ring rC5 = 0,(e,d,c,b,a),lp;
triangL(stdfglm(cyclic(5)));
```

D.7.3.2 triangLfak

Procedure from library `triang.lib` (see Section D.7.3 [triang_lib], page 559).

Usage: triangLfak(G); G=ideal

Assume: G is the reduced lexicographical Groebner bases of the zero-dimensional ideal (G), sorted by increasing leading terms.

Return: a list of finitely many triangular systems, such that the union of their varieties equals the variety of (G).

Note: Algorithm of Lazard with factorization (see: Lazard, D.: Solving zero-dimensional algebraic systems, J. Symb. Comp. 13, 117 - 132, 1992).

Remark: each polynomial of the triangular systems is factorized.

Example:

```
LIB "triang.lib";
ring rC5 = 0,(e,d,c,b,a),lp;
triangLfak(stdfglm(cyclic(5)));
```

D.7.3.3 triangM

Procedure from library `triang.lib` (see Section D.7.3 [triang_lib], page 559).

Usage: triangM(G[,i]); G=ideal, i=integer,

Assume: G is the reduced lexicographical Groebner bases of the zero-dimensional ideal (G), sorted by increasing leading terms.

Return: a list of finitely many triangular systems, such that the union of their varieties equals the variety of (G). If i = 2, then each polynomial of the triangular systems is factorized.

Note: Algorithm of Moeller (see: Moeller, H.M.:
On decomposing systems of polynomial equations with
finitely many solutions, Appl. Algebra Eng. Commun. Comput. 4, 217 - 230, 1993).

Example:

```
LIB "triang.lib";
ring rC5 = 0,(e,d,c,b,a),lp;
triangM(stdfglm(cyclic(5))); //oder: triangM(stdfglm(cyclic(5)),2);
```

D.7.3.4 triangMH

Procedure from library `triang.lib` (see Section D.7.3 [triang_lib], page 559).

Usage: triangMH(G[,i]); G=ideal, i=integer

Assume: G is the reduced lexicographical Groebner bases of the zero-dimensional ideal (G), sorted by increasing leading terms.

Return: a list of finitely many triangular systems, such that the disjoint union of their varieties equals the variety of (G). If i = 2, then each polynomial of the triangular systems is factorized.

Note: Algorithm of Moeller and Hillebrand (see: Moeller, H.M.: On decomposing systems of polynomial equations with finitely many solutions, Appl. Algebra Eng. Commun. Comput. 4, 217 - 230, 1993 and Hillebrand, D.: Triangulierung nulldimensionaler Ideale - Implementierung und Vergleich zweier Algorithmen, master thesis, Universitaet Dortmund, Fachbereich Mathematik, Prof. Dr. H.M. Moeller, 1999).

Example:

```
LIB "triang.lib";
ring rC5 = 0,(e,d,c,b,a),lp;
triangMH(stdfglm(cyclic(5)));
```

D.7.4 ntsolve_lib

Library: ntsolve.lib

Purpose: Real Newton Solving of Polynomial Systems

Authors: Wilfred Pohl, email: pohl@mathematik.uni-kl.de
Dietmar Hillebrand

Procedures:

D.7.4.1 nt_solve

Procedure from library `ntsolve.lib` (see Section D.7.4 [ntsolve_lib], page 561).

Usage: nt_solve(gls,ini[,ipar]); gls,ini= ideals, ipar=list/intvec,
gls: contains the equations, for which a solution will be computed ini: ideal of initial values (approximate solutions to start with),
ipar: control integers (default: ipar = 100,10)
ipar[1]: max. number of iterations
ipar[2]: accuracy (we have the l_2-norm `||.||`): accept solution `sol`
if `||`gls(sol)`||` `<` eps0*(0.1^ipar[2])
where eps0 = `||`gls(ini)`||` is the initial error

Assume: gls is a zerodimensional ideal with nvars(basering) = size(gls) (`>`1)

Return: ideal, coordinates of one solution (if found), 0 else

Note: if printlevel `>`0: displays comments (default =0)

Example:

```
LIB "ntsolve.lib";
ring rsq = (real,40),(x,y,z,w),lp;
ideal gls =  x2+y2+z2-10, y2+z3+w-8, xy+yz+xz+w5 - 1,w3+y;
ideal ini = 3.1,2.9,1.1,0.5;
intvec ipar = 200,0;
ideal sol = nt_solve(gls,ini,ipar);
sol;
↦ sol[1]=0.8698104581550055082008024750939710335537
↦ sol[2]=2.8215774457503246008496262517717182369409
↦ sol[3]=1.1323120084664179900060940157112668717318
↦ sol[4]=-1.4130710264066788493979994755901942396628
```

D.7.4.2 triMNewton

Procedure from library `ntsolve.lib` (see Section D.7.4 [ntsolve_lib], page 561).

Usage: triMNewton(G,a[,ipar]); G,a= ideals, ipar=list/intvec

Assume: G: g1,..,gn, a triangular system of n equations in n vars, i.e. gi=gi(var(n-i+1),..,var(n)),
a: ideal of numbers, coordinates of an approximation of a common zero of G to start with (with a[i] to be substituted in var(i)),
ipar: control integer vector (default: ipar = 100,10)
ipar[1]: max. number of iterations
ipar[2]: accuracy (we have as norm `|.|` absolute value):
accept solution `sol` if `|`G(sol)`|` `<` `|`G(a)`|`*(0.1^ipar[2]).

Return: an ideal, coordinates of a better approximation of a zero of G

Example:

```
LIB "ntsolve.lib";
ring r = (real,30),(z,y,x),(lp);
ideal i = x^2-1,y^2+x4-3,z2-y4+x-1;
ideal a = 2,3,4;
intvec e = 20,10;
ideal l = triMNewton(i,a,e);
l;
↦ l[1]=-2.000000000042265738880279143423
↦ l[2]=1.41421356237309504880168872421
↦ l[3]=1
```

D.7.5 zeroset_lib

Library: zeroset.lib

Purpose: Procedures For Roots and Factorization

Author: Thomas Bayer, email: tbayer@mathematik.uni-kl.de http://wwwmayr.informatik.tu-muenchen.de/personen/bayert/ Current Adress: Institut fuer Informatik, TU Muenchen

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

Procedures: Auxiliary procedures:

D.7.5.1 EGCD

Procedure from library `zeroset.lib` (see Section D.7.5 [zeroset_lib], page 562).

Usage: EGCD(f, g); where f,g are polynomials

Purpose: compute the polynomial gcd of f and g over Q(a)[x]

Return: polynomial h s.t. h is a greatest common divisor of f and g (not nec. monic)

Assume: basering = Q(a)[t]

Example:

```
LIB "zeroset.lib";
ring R = (0,a), x, lp;
minpoly = a2+1;
poly f =  x4 - 1;
poly g = x2 - 2*a*x - 1;
EGCD(f, g);
↦ (-4a)*x-4
```

D.7.5.2 Factor

Procedure from library `zeroset.lib` (see Section D.7.5 [zeroset_lib], page 562).

Usage: Factor(f); where f is a polynomial

Purpose: compute the factorization of the squarefree poly f over Q(a)[t]

Return: list with two entries
_[1] = factors (monic), first entry is the leading coefficient
_[2] = multiplicities (not yet implemented)

Assume: basering must be the univariate polynomial ring over a field, which is Q or a simple extension of Q given by a minpoly.

Note: if basering = Q[t] then this is the built-in `factorize`

Example:

```
LIB "zeroset.lib";
ring R = (0,a), x, lp;
minpoly = a2+1;
poly f =  x4 - 1;
list fl = Factor(f);
fl;
↦ [1]:
↦    _[1]=1
↦    _[2]=(40a+60)*x+(40a+60)
↦    _[3]=(9/65a-7/65)*x+(-7/65a-9/65)
↦    _[4]=(5a+1)*x+(-a+5)
↦    _[5]=(7/520a-1/130)*x+(-7/520a+1/130)
↦ [2]:
↦    _[1]=1
↦    _[2]=1
↦    _[3]=1
↦    _[4]=1
↦    _[5]=1
fl[1][1]*fl[1][2]*fl[1][3]*fl[1][4]*fl[1][5] - f;
↦ 0
```

D.7.5.3 Quotient

Procedure from library `zeroset.lib` (see Section D.7.5 [zeroset_lib], page 562).

Usage: Quotient(f, g); where f,g are polynomials;

Purpose: compute the quotient q and remainder r s.t. f = g*q + r, deg(r) < deg(g)

Return: list of polynomials
_[1] = quotient q
_[2] = remainder r

Assume: basering = Q[x] or Q(a)[x]

Example:

```
LIB "zeroset.lib";
ring R = (0,a), x, lp;
minpoly = a2+1;
poly f =  x4 - 2;
poly g = x - a;
list qr = Quotient(f, g);
qr;
↦ [1]:
↦    x3+(a)*x2-x+(-a)
↦ [2]:
↦    0
qr[1]*g + qr[2] - f;
↦ 1
```

D.7.5.4 Remainder

Procedure from library `zeroset.lib` (see Section D.7.5 [zeroset_lib], page 562).

Usage: Remainder(f, g); where f,g are polynomials

Purpose: compute the remainder of the division of f by g, i.e. a polynomial r s.t. f = g*q + r, deg(r) < deg(g).

Return: poly

Assume: basering = Q[x] or Q(a)[x]

Example:

```
LIB "zeroset.lib";
ring R = (0,a), x, lp;
minpoly = a2+1;
poly f =  x4 - 1;
poly g = x3 - 1;
Remainder(f, g);
↦ x-1
```

D.7.5.5 Roots

Procedure from library `zeroset.lib` (see Section D.7.5 [zeroset_lib], page 562).

Usage: Roots(f); where f is a polynomial

Purpose: compute all roots of f in a finite extension of the ground field without multiplicities.

Return: ring, a polynomial ring over an extension field of the ground field, containing a list 'roots' and polynomials 'newA' and 'f':

- 'roots' is the list of roots of the polynomial f (no multiplicities)
- if the ground field is Q(a') and the extension field is Q(a), then
  'newA' is the representation of a' in Q(a).
  If the basering contains a parameter 'a' and the minpoly remains unchanged
  then 'newA' = 'a'.
  If the basering does not contain a parameter then 'newA' = 'a' (default).
- 'f' is the polynomial f in Q(a) (a' being substituted by 'newA')

Assume: ground field to be Q or a simple extension of Q given by a minpoly

Example:

```
LIB "zeroset.lib";
ring R = (0,a), x, lp;
minpoly = a2+1;
poly f = x3 - a;
def R1 = Roots(f);
↦
↦ // 'Roots' created a new ring which contains the list 'roots' and
↦ // the polynomials 'f' and 'newA'
↦ // To access the roots, newA and the new representation of f, type
↦    def R = Roots(f); setring R; roots; newA; f;
↦
setring R1;
minpoly;
↦ (a4-a2+1)
newA;
↦ (a3)
f;
↦ x3+(-a3)
roots;
↦ [1]:
↦    (-a3)
↦ [2]:
↦    (a)
↦ [3]:
↦    (a3-a)
map F;
F[1] = roots[1];
F(f);
↦ 0
```

D.7.5.6 SQFRNorm

Procedure from library `zeroset.lib` (see Section D.7.5 [zeroset_lib], page 562).

Usage: SQFRNorm(f); where f is a polynomial

Purpose: compute the norm of the squarefree polynomial f in Q(a)[x].

Return: list with 3 entries
_[1] = squarefree norm of g (poly)
_[2] = g (= f(x - s*a)) (poly)
_[3] = s (int)

Assume: f must be squarefree, basering = Q(a)[x] and minpoly != 0.

Note: the norm is an element of Q[x]

Example:

```
LIB "zeroset.lib";
ring R = (0,a), x, lp;
minpoly = a2+1;
poly f =  x4 - 2*x + 1;
SQFRNorm(f);
↦ [1]:
↦    x8+4*x6-4*x5+8*x4+8*x3-4*x2+8*x+8
```

```
↦ [2]:
↦    x4+(-4a)*x3-6*x2+(4a-2)*x+(2a+2)
↦ [3]:
↦    1
```

D.7.5.7 ZeroSet

Procedure from library `zeroset.lib` (see Section D.7.5 [zeroset_lib], page 562).

Usage: ZeroSet(I [,opt]); I=ideal, opt=integer

Purpose: compute the zero-set of the zero-dim. ideal I, in a finite extension of the ground field.

Return: ring, a polynomial ring over an extension field of the ground field, containing a list 'zeroset', a polynomial 'newA', and an ideal 'id':
- 'zeroset' is the list of the zeros of the ideal I, each zero is an ideal.
- if the ground field is Q(a') and the extension field is Q(a), then
  'newA' is the representation of a' in Q(a).
  If the basering contains a parameter 'a' and the minpoly remains unchanged
  then 'newA' = 'a'.
  If the basering does not contain a parameter then 'newA' = 'a' (default).
- 'id' is the ideal I in Q(a)[x_1,...] (a' substituted by 'newA')

Assume: dim(I) = 0, and ground field to be Q or a simple extension of Q given by a minpoly.

Options: opt = 0 no primary decomposition (default)
opt > 0 primary decomposition

Note: If I contains an algebraic number (parameter) then 'I' must be transformed w.r.t. 'newA' in the new ring.

Example:

```
LIB "zeroset.lib";
ring R = (0,a), (x,y,z), lp;
minpoly = a2 + 1;
ideal I = x2 - 1/2, a*z - 1, y - 2;
def T = ZeroSet(I);
↦ 1
setring T;
minpoly;
↦ (4a4+4a2+9)
newA;
↦ (1/3a3+5/6a)
id;
↦ id[1]=(1/3a3+5/6a)*z-1
↦ id[2]=y-2
↦ id[3]=2*x2-1
zeroset;
↦ [1]:
↦    _[1]=(-1/3a3+1/6a)
↦    _[2]=2
↦    _[3]=(-1/3a3-5/6a)
↦ [2]:
↦    _[1]=(1/3a3-1/6a)
↦    _[2]=2
```

```
↦    _[3]=(-1/3a3-5/6a)
map F1 = basering, zeroset[1];
map F2 = basering, zeroset[2];
F1(id);
↦ _[1]=0
↦ _[2]=0
↦ _[3]=0
F2(id);
↦ _[1]=0
↦ _[2]=0
↦ _[3]=0
```

D.7.5.8 EGCDMain

Procedure from library `zeroset.lib` (see Section D.7.5 [zeroset_lib], page 562).

Purpose: compute the polynomial gcd of f and g over Q(a)[x]

Return: poly

Assume: basering = Q[x,a] and ideal mpoly is defined (it might be 0), this represents the ring Q(a)[x] together with its minimal polynomial.

D.7.5.9 FactorMain

Procedure from library `zeroset.lib` (see Section D.7.5 [zeroset_lib], page 562).

Purpose: compute the factorization of the squarefree poly f over Q(a)[t], minpoly = p(a).

Return: list with 2 entries
  _[1] = factors, first is a constant
  _[2] = multiplicities (not yet implemented)

Assume: basering = Q[x,a], representing Q(a)[x]. An ideal mpoly must be defined, representing the minimal polynomial (it might be 0!).

D.7.5.10 InvertNumberMain

Procedure from library `zeroset.lib` (see Section D.7.5 [zeroset_lib], page 562).

Usage: InvertNumberMain(f); where f is a polynomial

Purpose: compute 1/f if f is a number in Q(a) i.e., f is represented by a polynomial in Q[a].

Return: poly 1/f

Assume: basering = Q[x_1,...,x_n,a], ideal mpoly must be defined and != 0 !

D.7.5.11 QuotientMain

Procedure from library `zeroset.lib` (see Section D.7.5 [zeroset_lib], page 562).

Usage: QuotientMain(f, g); where f,g are polynomials

Purpose: compute the quotient q and remainder r s.t. f = g*q + r, deg(r) < deg(g)

Return: list of polynomials

_[1] = quotient q
_[2] = remainder r

Assume: basering = Q[x,a] and ideal mpoly is defined (it might be 0), this represents the ring Q(a)[x] together with its minimal polynomial.

D.7.5.12 RemainderMain

Procedure from library `zeroset.lib` (see Section D.7.5 [zeroset_lib], page 562).

Usage: RemainderMain(f, g); where f,g are polynomials

Purpose: compute the remainder r s.t. f = g*q + r, deg(r) < deg(g)

Return: poly

Assume: basering = Q[x,a] and ideal mpoly is defined (it might be 0), this represents the ring Q(a)[x] together with its minimal polynomial.

D.7.5.13 RootsMain

Procedure from library `zeroset.lib` (see Section D.7.5 [zeroset_lib], page 562).

Usage: RootsMain(f); where f is a polynomial

Purpose: compute all roots of f in a finite extension of the ground field without multiplicities.

Return: list, all entries are polynomials
_[1] = roots of f, each entry is a polynomial
_[2] = 'newA' - if the ground field is Q(a') and the extension field is Q(a), then 'newA' is the representation of a' in Q(a)
_[3] = minpoly of the algebraic extension of the ground field

Assume: basering = Q[x,a] ideal mpoly must be defined, it might be 0!

Note: might change the ideal mpoly !!

D.7.5.14 SQFRNormMain

Procedure from library `zeroset.lib` (see Section D.7.5 [zeroset_lib], page 562).

Usage: SQFRNorm(f); where f is a polynomial

Purpose: compute the norm of the squarefree polynomial f in Q(a)[x].

Return: list with 3 entries
_[1] = squarefree norm of g (poly)
_[2] = g (= f(x - s*a)) (poly)
_[3] = s (int)

Assume: f must be squarefree, basering = Q[x,a] and ideal mpoly is equal to 'minpoly',this represents the ring Q(a)[x] together with 'minpoly'.

Note: the norm is an element of Q[x]

D.7.5.15 ContainedQ

Procedure from library `zeroset.lib` (see Section D.7.5 [zeroset_lib], page 562).

Usage: ContainedQ(data, f [, opt]); data=list; f=any type, opt=integer

Purpose: test if f is an element of data.

Return: int
0 if f not contained in data
1 if f contained in data

Options: opt = 0 : use '==' for comparing f with elements from data
opt = 1 : use `SameQ` for comparing f with elements from data

D.7.5.16 SameQ

Procedure from library `zeroset.lib` (see Section D.7.5 [zeroset_lib], page 562).

Usage: SameQ(a, b); a,b=list/intvec

Purpose: test a == b elementwise, i.e., a[i] = b[i].

Return: int
0 if a != b
1 if a == b

D.8 Visualization

D.8.1 graphics_lib

Library: graphics.lib

Purpose: Procedures to use Graphics with Mathematica

Author: Christian Gorzel, gorzelc@math.uni-muenster.de

Procedures:

D.8.1.1 staircase

Procedure from library `graphics.lib` (see Section D.8.1 [graphics_lib], page 569).

Usage: staircase(s,I); s a string, I ideal in two variables

Return: string with Mathematica input for displaying staircase diagrams of an ideal I, i.e. exponent vectors of the initial ideal of I

Note: ideal I should be given by a standard basis. Let s="" and copy and paste the result into a Mathematica notebook.

Example:

```
LIB "graphics.lib";
ring r0 = 0,(x,y),ls;
ideal I = -1x2y6-1x4y2, 7x6y5+1/2x7y4+6x4y6;
staircase("",std(I));
ring r1 = 0,(x,y),dp;
ideal I = fetch(r0,I);
```

```
staircase("",std(I));
ring r2 = 0,(x,y),wp(2,3);
ideal I = fetch(r0,I);
staircase("",std(I));
// Paste the output into a Mathematica notebook
// active evalutation of the cell with SHIFT RETURN
```

D.8.1.2 mathinit

Procedure from library `graphics.lib` (see Section D.8.1 [graphics_lib], page 569).

Usage: mathinit();

Return: initializing string for loading Mathematica's ImplicitPlot

Example:

```
LIB "graphics.lib";
mathinit();
// Paste the output into a Mathematica notebook
// active evalutation of the cell with SHIFT RETURN
```

D.8.1.3 mplot

Procedure from library `graphics.lib` (see Section D.8.1 [graphics_lib], page 569).

Usage: mplot(fname, I [,I1,I2,..,s]); fname=string; I,I1,I2,..=ideals, s=string representing the plot region.
Use the ideals I1,I2,.. in order to produce multiple plots (they need to have the same number of entries as I!).

Return: string, text with Mathematica commands to display a plot

Note: The plotregion is defaulted to -1,1 around zero.
For implicit given curves enter first the string returned by proc mathinit into Mathematica in order to load ImplicitPlot. The following conventions for I are used:
- ideal with 2 entries in one variable means a parametrised plane curve,
- ideal with 3 entries in one variable means a parametrised space curve,
- ideal with 3 entries in two variables means a parametrised surface,
- ideal with 2 entries in two variables means an implicit curve
  given as I[1]==I[2],
- ideal with 1 entry (or one polynomial) in two variables means
  an implicit curve given as f == 0,

Example:

```
LIB "graphics.lib";
// ---------  plane curves ------------
ring rr0 = 0,x,dp; export rr0;
ideal I = x3 + x, x2;
ideal J = x2, -x+x3;
mplot("",I,J,"-2,2");
// Paste the output into a Mathematica notebook
// active evalutation of the cell with SHIFT RETURN
// --------- space curves --------------
I = x3,-1/10x3+x2,x2;
mplot("",I);
```

```
// Paste the output into a Mathematica notebook
// active evalutation of the cell with SHIFT RETURN
// ----------- surfaces -------------------
ring rr1 = 0,(x,y),dp; export rr1;
ideal J = xy,y,x2;
mplot("",J,"-2,1","1,2");
// Paste the output into a Mathematica notebook
// active evalutation of the cell with SHIFT RETURN
kill rr0,rr1;
```

D.8.2 latex_lib

Library: latex.lib

Purpose: Typesetting of Singular-Objects in LaTeX2e

Author: Christian Gorzel, gorzelc@math.uni-muenster.de

Procedures:

Global variables:

TeXwidth, TeXnofrac, TeXbrack, TeXproj, TeXaligned, TeXreplace, NoDollars are used to control the typesetting. Call `texdemo();` to obtain a LaTeX2e file `texlibdemo.tex` explaining the features of `latex.lib` and its global variables.

`TeXwidth` (int) -1, 0, 1..9, >9: controls breaking of long polynomials
`TeXnofrac` (int) flag: write 1/2 instead of \frac{1}{2}
`TeXbrack` (string) "{", "(", "<", "|", empty string:
controls brackets around ideals and matrices
`TeXproj` (int) flag: write ":" instead of "," in vectors
`TeXaligned` (int) flag: write maps (and ideals) aligned
`TeXreplace` (list) list entries = 2 strings: replacing symbols
`NoDollars` (int) flag: suppresses surrounding $ signs

D.8.2.1 closetex

Procedure from library `latex.lib` (see Section D.8.2 [latex_lib], page 571).

Usage: closetex(fname); fname string

Return: nothing; writes a LaTeX2e closing line into file `<fname>`.

Note: preceding ">>" are deleted and suffix ".tex" (if not given) is added to `fname`.

Example:

```
LIB "latex.lib";
opentex("exmpl");
texobj("exmpl","{\\large \\bf hello}");
closetex("exmpl");
```

D.8.2.2 opentex

Procedure from library `latex.lib` (see Section D.8.2 [latex_lib], page 571).

Usage: opentex(fname); fname string

Return: nothing; writes a LaTeX2e header into a new file `<fname>`.

Note: preceding ">>" are deleted and suffix ".tex" (if not given) is added to `fname`.

Example:

```
LIB "latex.lib";
opentex("exmpl");
texobj("exmpl","hello");
closetex("exmpl");
```

D.8.2.3 tex

Procedure from library `latex.lib` (see Section D.8.2 [latex_lib], page 571).

Usage: tex(fname); fname string

Return: nothing; calls latex (LaTeX2e) for compiling the file fname

Note: preceding ">>" are deleted and suffix ".tex" (if not given) is added to `fname`.

Example:

```
LIB "latex.lib";
ring r;
ideal I = maxideal(7);
opentex("exp001");              // open latex2e document
texobj("exp001","An ideal ",I);
closetex("exp001");
tex("exp001");
↦ calling  latex2e  for : exp001.tex
↦
↦ This is TeX, Version 3.14159 (C version 6.1)
↦ (exp001.tex
↦ LaTeX2e <1996/12/01> patch level 1
↦ Babel <v3.6h> and hyphenation patterns for american, german, loaded.
↦ (/usr/lib/texmf/texmf/tex/latex/base/article.cls
↦ Document Class: article 1996/10/31 v1.3u Standard LaTeX document class
↦ (/usr/lib/texmf/texmf/tex/latex/base/size10.clo))
↦ (/usr/lib/texmf/texmf/tex/latex/amslatex/amsmath.sty
↦ (/usr/lib/texmf/texmf/tex/latex/amslatex/amstext.sty
↦ (/usr/lib/texmf/texmf/tex/latex/amslatex/amsgen.sty))
↦ (/usr/lib/texmf/texmf/tex/latex/amslatex/amsbsy.sty)
↦ (/usr/lib/texmf/texmf/tex/latex/amslatex/amsopn.sty))
↦ (/usr/lib/texmf/texmf/tex/latex/amsfonts/amssymb.sty
↦ (/usr/lib/texmf/texmf/tex/latex/amsfonts/amsfonts.sty))
↦ No file exp001.aux.
↦ (/usr/lib/texmf/texmf/tex/latex/amsfonts/umsa.fd)
↦ (/usr/lib/texmf/texmf/tex/latex/amsfonts/umsb.fd) [1] (exp001.aux) )
↦ Output written on exp001.dvi (1 page, 2912 bytes).
↦ Transcript written on exp001.log.
system("sh","rm exp001.*");
↦ 0
```

D.8.2.4 texdemo

Procedure from library `latex.lib` (see Section D.8.2 [latex_lib], page 571).

Usage: texdemo();

Return: nothing; generates a LaTeX2e file called `texlibdemo.tex` explaining the features of `latex.lib` and its global variables.

Note: this proc may take some time.

D.8.2.5 texfactorize

Procedure from library `latex.lib` (see Section D.8.2 [latex_lib], page 571).

Usage: texfactorize(fname,f); fname string, f poly

Return: if `fname=""`: string, f as a product of its irreducible factors
otherwise: append this string to the file `<fname>`, and return nothing.

Note: preceding ">>" are deleted and suffix ".tex" (if not given) is added to `fname`.

Example:

```
LIB "latex.lib";
ring r2 = 13,(x,y),dp;
poly f = (x+1+y)^2*x3y*(2x-2y)*y12;
texfactorize("",f);
↦ $-2\cdot x^{3}\cdot y^{13}\cdot (-x+y)\cdot (x+y+1)^{2}$
ring R49 = (7,a),x,dp;
minpoly = a2+a+3;
poly f = (a24x5+x3)*a2x6*(x+1)^2;
f;
↦ (a+3)*x13+(2a-1)*x12+(-2a+1)*x10+(-a-3)*x9
texfactorize("",f);
↦ $(a+3)\cdot (x-1)\cdot (x+1)^{3}\cdot x^{9}$
```

D.8.2.6 texmap

Procedure from library `latex.lib` (see Section D.8.2 [latex_lib], page 571).

Usage: texmap(fname,m,@r1,@r2); fname string, m string/map, @r1,@r2 rings

Return: if `fname=""`: string, the map m from @r1 to @r2 (preceded by its name if m = string) in TeX-typesetting;
otherwise: append this string to the file `<fname>`, and return nothing.

Note: preceding ">>" are deleted in `fname`, and suffix ".tex" (if not given) is added to `fname`.
If m is a string then it has to be the name of an existing map from @r1 to @r2.

Example:

```
LIB "latex.lib";
// -------- prepare for example ---------
if (defined(TeXaligned)) {int Teali=TeXaligned; kill TeXaligned;}
if (defined(TeXreplace)) {list Terep=TeXreplace; kill TeXreplace;}
// -------- the example starts here ---------
//
string fname = "tldemo";
ring @r1=0,(x,y,z),dp;
if(system("with","Namespaces")) { exportto(Current, @r1); }
else { export @r1; }
↦ // ** `@r1` is already global
ring r2=0,(u,v),dp;
map @phi =(@r1,u2,uv -v,v2); export @phi;
↦ // ** `@phi` is already global
list TeXreplace;
```

```
TeXreplace[1] = list("@phi","\\phi");    // @phi --> \phi
export TeXreplace;
↦ // ** ‘TeXreplace‘ is already global
texmap("","@phi",@r1,r2);                // standard form
↦ $$
↦ \begin{array}{rcc}
↦ \phi:\Q[x,y,z] & \longrightarrow & \Q[u,v]\\[2mm]
↦ \left(x,y,z\right) & \longmapsto &
↦  \left(
↦ \begin{array}{c}
↦ u^{2}\\
↦ uv-v\\
↦ v^{2}
↦ \end{array}
↦ \right)
↦ \end{array}
↦ $$
//
int TeXaligned; export TeXaligned;       // map in one line
↦ // ** ‘TeXaligned‘ is already global
texmap("",@phi,@r1,r2);
↦ $\Q[x,y,z]\longrightarrow\Q[u,v], \ \left(x,y,z\right)\longmapsto \left(u\
   ^{2},uv-v,v^{2}\right)$
//
kill @r1,TeXreplace,TeXaligned;
//
// --- restore global variables if previously defined ---
if (defined(Teali)) {int TeXaligned=Teali; export TeXaligned; kill Teali;}
if (defined(Terep)) {list TeXreplace=Terep; export TeXreplace; kill Terep;}
```

D.8.2.7 texname

Procedure from library `latex.lib` (see Section D.8.2 [latex_lib], page 571).

Usage: texname(fname,s); fname,s strings

Return: if `fname=""`: string, the transformed string s, where the following rules apply:

```
              s' + "~"             -->  "\\tilde{"+ s' +"}"
             "_" + int             -->       "_{" + int +"}"
          "[" + s' + "]"           -->      "_{" + s' + "}"
           "A..Z" + int            --> "A..Z" + "^{" + int + "}"
           "a..z" + int            --> "a..z" + "_{" + int + "}"
        "(" + int + "," + s' + ")" --> "_{"+ int +"}" + "^{" + s'+"}"
```

Anyhow, strings which begin with a `"{"` are only changed by deleting the first and last character (intended to remove the surrounding curly brackets).

if `fname!=""`: append the transformed string s to the file `<fname>`, and return nothing.

Note: preceding `">>"` are deleted in `fname`, and suffix `".tex"` (if not given) is added to `fname`.

Example:

```
LIB "latex.lib";
ring r = 0,(x,y),lp;
poly f = 3xy4 + 2xy2 + x5y3 + x + y6;
texname("","{f(10)}");
```

```
↦ f(10)
texname("","f(10) =");
↦ f_{10} =
texname("","n1");
↦ n_{1}
texname("","T1_12");
↦ T^{1}_{12}
texname("","g'_11");
↦ g'_{11}
texname("","f23");
↦ f_{23}
texname("","M[2,3]");
↦ M_{2,3}
texname("","A(0,3);");
↦ A_{0}^{3};
texname("","E~(3)");
↦ \tilde{E}_{3}
```

D.8.2.8 texobj

Procedure from library `latex.lib` (see Section D.8.2 [latex_lib], page 571).

Usage: texobj(fname,l); fname string, l list

Return: if `fname=""`: string, the entries of l in LaTeX-typesetting;
otherwise: append this string to the file `<fname>`, and return nothing.

Note: preceding "`>>`" are deleted in `fname`, and suffix ".tex" (if not given) is added to `fname`.

Example:

```
LIB "latex.lib";
// -------- prepare for example ---------
if (defined(TeXaligned)) {int Teali=TeXaligned; kill TeXaligned;}
if (defined(TeXbrack)){string Tebra=TeXbrack; kill TeXbrack;}
//
//  --------------  typesetting for polynomials ----------
ring r = 0,(x,y),lp;
poly f = x5y3 + 3xy4 + 2xy2 + y6;
f;
↦ x5y3+3xy4+2xy2+y6
texobj("",f);
↦ $$\begin{array}{rl}
↦ & x^{5}y^{3}+3xy^{4}+2xy^{2}+y^{6}\\
↦ \end{array}
↦ $$
↦
//  --------------  typesetting for ideals ----------
ideal G = jacob(f);
G;
↦ G[1]=5x4y3+3y4+2y2
↦ G[2]=3x5y2+12xy3+4xy+6y5
texobj("",G);
↦ $$\left(
↦ \begin{array}{c}
↦ 5x^{4}y^{3}+3y^{4}+2y^{2}, \\
```

```
↦ 3x^{5}y^{2}+12xy^{3}+4xy+6y^{5}
↦ \end{array}
↦ \right)$$
↦
//  --------------  variation of typesetting for ideals ----------
int TeXaligned = 1; export TeXaligned;
↦ // ** ‘TeXaligned‘ is already global
string TeXbrack = "<"; export TeXbrack;
↦ // ** ‘TeXbrack‘ is already global
texobj("",G);
↦ $\left<5x^{4}y^{3}+3y^{4}+2y^{2},3x^{5}y^{2}+12xy^{3}+4xy+6y^{5}\right>$
↦
kill TeXaligned, TeXbrack;
//  --------------  typesetting for matrices ----------
matrix J = jacob(G);
texobj("",J);
↦ $$\left(
↦ \begin{array}{*{2}{c}}
↦ 20x^{3}y^{3} & 15x^{4}y^{2}+12y^{3}+4y \\
↦ 15x^{4}y^{2}+12y^{3}+4y & 6x^{5}y+36xy^{2}+4x+30y^{4}
↦ \end{array}
↦ \right)
↦ $$
↦
//  --------------  typesetting for intmats ----------
intmat m[3][4] = 9,2,4,5,2,5,-2,4,-6,10,-1,2,7;
texobj("",m);
↦ $$\left(
↦ \begin{array}{*{4}{r}}
↦ 9 & 2 & 4 & 5\\
↦ 2 & 5 & -2 & 4\\
↦ -6 & 10 & -1 & 2
↦ \end{array}
↦ \right)
↦ $$
↦
//
// --- restore global variables if previously defined ---
if (defined(Teali)){int TeXaligned=Teali; export TeXaligned; kill Teali;}
if (defined(Tebra)){string TeXbrack=Tebra; export TeXbrack; kill Tebra;}
```

D.8.2.9 texpoly

Procedure from library `latex.lib` (see Section D.8.2 [latex_lib], page 571).

Usage: texpoly(fname,p); fname string, p poly

Return: if `fname=""`: string, the poly p in LaTeX-typesetting;
otherwise: append this string to the file `<fname>`, and return nothing.

Note: preceding ">>" are deleted in `fname`, and suffix ".tex" (if not given) is added to `fname`.

Example:

```
LIB "latex.lib";
ring r0=0,(x,y,z),dp;
```

```
poly f = -1x^2 + 2;
texpoly("",f);
↦ $-x^{2}+2$
ring rr= real,(x,y,z),dp;
texpoly("",2x2y23z);
↦ $2.000x^{2}y^{23}z$
ring r7= 7,(x,y,z),dp;
poly f = 2x2y23z;
texpoly("",f);
↦ $2x^{2}y^{23}z$
ring rab =(0,a,b),(x,y,z),dp;
poly f = (-2a2 +b3 -2)/a * x2y4z5 + (a2+1)*x + a+1;
f;
↦ (-2a2+b3-2)/(a)*x2y4z5+(a2+1)*x+(a+1)
texpoly("",f);
↦ $-\frac{2a^{2}-b^{3}+2}{a}x^{2}y^{4}z^{5}+(a^{2}+1)x+(a+1)$
```

D.8.2.10 texproc

Procedure from library `latex.lib` (see Section D.8.2 [latex_lib], page 571).

Usage: texproc(fname,pname); fname,pname strings

Assume: ‘`pname`‘ is a procedure.

Return: if `fname=""`: string, the proc ‘`pname`‘ in a verbatim environment in LaTeX-typesetting; otherwise: append this string to the file `<fname>`, and return nothing.

Note: preceding ">>" are deleted in `fname`, and suffix ".tex" (if not given) is added to `fname`. `texproc` cannot be applied to itself correctly.

Example:

```
LIB "latex.lib";
proc exp(int i,int j,list #)
{ string s;
if (size(#))
{
for(i;i<=j;i++)
{ s = s + string(j) + string(#); }
}
return(s);
}
export exp;
↦ // ** ‘exp‘ is already global
texproc("","exp");
↦ \begin{verbatim}
↦ proc exp(int i,int j,list #)
↦ {
↦  string s;
↦ if (size(#))
↦ {
↦ for(i;i<=j;i++)
↦ { s = s + string(j) + string(#); }
↦ }
↦ return(s);
```

```
↦
↦ }
↦ \end{verbatim}
↦
kill exp;
```

D.8.2.11 texring

Procedure from library `latex.lib` (see Section D.8.2 [latex_lib], page 571).

Usage: texring(fname, r[,L]); fname string, r ring, L list

Return: if `fname=""`: string, the ring in TeX-typesetting;
otherwise: append this string to the file `<fname>` and return nothing.

Note: preceding `">>"` are deleted and suffix ".tex" (if not given) is added to `fname`.
The optional list L is assumed to be a list of strings which control, e.g., the symbol for the field of coefficients.
For more details call `texdemo();` (generates a LaTeX2e file called `texlibdemo.tex` which explains all features of `texring`).

Example:

```
LIB "latex.lib";
ring r0 = 0,(x,y),dp;                // char = 0, polynomial ordering
texring("",r0);
↦ $\Q[x,y]$
//
ring r7 =7,(x(0..2)),ds;             // char = 7, local ordering
texring("",r7);
↦ $\Z_{7}[[x_{0},x_{1},x_{2}]]$
//
ring r1 = 0,(x1,x2,y1,y2),wp(1,2,3,4);
texring("",r1);
↦ $\Q[x_{1},x_{2},y_{1},y_{2}]$
//
ring rr = real,(x),dp;               // real numbers
texring("",rr);
↦ $\R[x]$
//
ring rabc =(0,t1,t2,t3),(x,y),dp;    // ring with parameters
texring("",rabc);
↦ $\Q(t_{1},t_{2},t_{3})[x,y]$
//
ring ralg = (7,a),(x1,x2),ds;        // algebraic extension
minpoly = a2-a+3;
texring("",ralg);
↦ $\Z_{7}(a)[[x_{1},x_{2}]]$
texring("",ralg,"mipo");
↦ $\Z_{7}(a)/(a^{2}-a+3)[[x_{1},x_{2}]]$
//
ring r49=(49,a),x,dp;                // Galois field
texring("",r49);
↦ $\F_{49}[x]$
//
setring r0;                          // quotient ring
```

```
ideal i = x2-y3;
qring q = std(i);
texring("",q);
↦ $\Q[x,y]/\left(y^{3}-x^{2}\right)
↦ $
//
// ------------------ additional features -------------------
ring r9 =0,(x(0..9)),ds;
texring("",r9,1);
↦ $\Q[[x_{0},\ldots,x_{9}]]$
texring("",r9,"C","{","^G");
↦ $\C\{x_{0},x_{1},x_{2},x_{3},x_{4},x_{5},x_{6},x_{7},x_{8},x_{9}\}^G$
//
ring rxy = 0,(x(1..5),y(1..6)),ds;
intvec v = 5,6;
texring("",rxy,v);
↦ $\Q[[x_{1},\ldots,x_{5},y_{1},\ldots,y_{6}]]$
```

D.8.2.12 rmx

Procedure from library `latex.lib` (see Section D.8.2 [latex_lib], page 571).

Usage: rmx(fname); fname string

Return: nothing; removes the `.log` and `.aux` files associated to the LaTeX file <fname>.

Note: If `fname` ends by `".dvi"` or `".tex"`, the `.dvi` or `.tex` file will be deleted, too.

Example:

```
LIB "latex.lib";
ring r;
poly f = x+y+z;
opentex("exp001");              // defaulted latex2e document
texobj("exp001","A polynom",f);
closetex("exp001");
tex("exp001");
↦ calling  latex2e  for : exp001.tex
↦
↦ This is TeX, Version 3.14159 (C version 6.1)
↦ (exp001.tex
↦ LaTeX2e <1996/12/01> patch level 1
↦ Babel <v3.6h> and hyphenation patterns for american, german, loaded.
↦ (/usr/lib/texmf/texmf/tex/latex/base/article.cls
↦ Document Class: article 1996/10/31 v1.3u Standard LaTeX document class
↦ (/usr/lib/texmf/texmf/tex/latex/base/size10.clo))
↦ (/usr/lib/texmf/texmf/tex/latex/amslatex/amsmath.sty
↦ (/usr/lib/texmf/texmf/tex/latex/amslatex/amstext.sty
↦ (/usr/lib/texmf/texmf/tex/latex/amslatex/amsgen.sty))
↦ (/usr/lib/texmf/texmf/tex/latex/amslatex/amsbsy.sty)
↦ (/usr/lib/texmf/texmf/tex/latex/amslatex/amsopn.sty))
↦ (/usr/lib/texmf/texmf/tex/latex/amsfonts/amssymb.sty
↦ (/usr/lib/texmf/texmf/tex/latex/amsfonts/amsfonts.sty))
↦ No file exp001.aux.
↦ (/usr/lib/texmf/texmf/tex/latex/amsfonts/umsa.fd)
```

```
↦ (/usr/lib/texmf/texmf/tex/latex/amsfonts/umsb.fd) [1] (exp001.aux) )
↦ Output written on exp001.dvi (1 page, 308 bytes).
↦ Transcript written on exp001.log.
rmx("exp001");   // removes aux and log file of exp001
system("sh","rm exp001.*");
↦ 0
```

D.8.2.13 xdvi

Procedure from library latex.lib (see Section D.8.2 [latex_lib], page 571).

Usage: xdvi(fname[,style]); fname,style = string

Return: nothing; displays dvi-file fname.dvi with previewer xdvi

Note: ending .dvi may miss in fname
style overwrites the default setting xdvi

Example:

```
LIB "latex.lib";
intmat m[3][4] = 9,2,4,5,2,5,-2,4,-6,10,-1,2,7;
opentex("exp001");
texobj("exp001","An intmat:  ",m);
closetex("exp001");
tex("exp001");
↦ calling  latex2e  for : exp001.tex
↦
↦ This is TeX, Version 3.14159 (C version 6.1)
↦ (exp001.tex
↦ LaTeX2e <1996/12/01> patch level 1
↦ Babel <v3.6h> and hyphenation patterns for american, german, loaded.
↦ (/usr/lib/texmf/texmf/tex/latex/base/article.cls
↦ Document Class: article 1996/10/31 v1.3u Standard LaTeX document class
↦ (/usr/lib/texmf/texmf/tex/latex/base/size10.clo))
↦ (/usr/lib/texmf/texmf/tex/latex/amslatex/amsmath.sty
↦ (/usr/lib/texmf/texmf/tex/latex/amslatex/amstext.sty
↦ (/usr/lib/texmf/texmf/tex/latex/amslatex/amsgen.sty))
↦ (/usr/lib/texmf/texmf/tex/latex/amslatex/amsbsy.sty)
↦ (/usr/lib/texmf/texmf/tex/latex/amslatex/amsopn.sty))
↦ (/usr/lib/texmf/texmf/tex/latex/amsfonts/amssymb.sty
↦ (/usr/lib/texmf/texmf/tex/latex/amsfonts/amsfonts.sty))
↦ No file exp001.aux.
↦ (/usr/lib/texmf/texmf/tex/latex/amsfonts/umsa.fd)
↦ (/usr/lib/texmf/texmf/tex/latex/amsfonts/umsb.fd) [1] (exp001.aux) )
↦ Output written on exp001.dvi (1 page, 524 bytes).
↦ Transcript written on exp001.log.
xdvi("exp001");
↦ calling  xdvi  for : exp001
↦
system("sh","rm exp001.*");
↦ 0
```

D.8.3 paramet_lib

Library: paramet.lib

Purpose: Parametrization of Varieties

Author: Thomas Keilen, keilen@mathematik.uni-kl.de

Procedures:

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

See also: Section D.5.5 [hnoether_lib], page 454; Section D.4.6 [normal_lib], page 406; Section D.4.7 [primdec_lib], page 409.

D.8.3.1 parametrize

Procedure from library `paramet.lib` (see Section D.8.3 [paramet_lib], page 580).

Usage: parametrize(I); I ideal in an arbitrary number of variables, whose radical is prime, in a ring with global ordering

Create: If the parametrization is successful, the basering will be changed to the parametrization ring, that is to the ring PR=0,(s,t),dp; respectively PR=0,t(1..d),dp;, depending on the dimension of the parametrized variety.

Return: a list containing the parametrization ideal resp. the original ideal, the number of variables needed for the parametrization resp. 0, and 1 resp. 0 depending on whether the parametrization was successful or not

Example:

```
LIB "paramet.lib";
ring RING=0,(x,y,z),dp;
ideal I=z2-y2x2+x3;
parametrize(I);
↦ [1]:
↦    _[1]=s2-t2
↦    _[2]=s
↦    _[3]=s2t-t3
↦ [2]:
↦    2
↦ [3]:
↦    1
```

See also: Section D.4.6.1 [normal], page 407; Section D.8.3.2 [parametrizepd], page 581; Section D.4.7.2 [primdecGTZ], page 410; Section D.4.7.7 [radical], page 412.

D.8.3.2 parametrizepd

Procedure from library `paramet.lib` (see Section D.8.3 [paramet_lib], page 580).

Usage: parametrizepd(I); I ideal in a polynomial ring with global ordering

Create: If the parametrization is successful, the basering will be changed to the parametrization ring, that is to the ring PR=0,(s,t),dp; respectively PR=0,t(1..d),dp;, depending on the dimension of the parametrized variety.

Return: a list of lists, where each entry contains the parametrization of a primary component of I resp. 0, the number of variables resp. 0, and 1 resp. 0 depending on whether the parametrization of the component was successful or not

Example:

```
LIB "paramet.lib";
ring RING=0,(x,y,z),dp;
ideal I=(x2-y2z2+z3)*(x2-z2-z3),(x2-y2z2+z3)*yz;
parametrizepd(I);
↦ [1]:
↦    [1]:
↦       _[1]=s2t-t3
↦       _[2]=s
↦       _[3]=s2-t2
↦    [2]:
↦       2
↦    [3]:
↦       1
↦ [2]:
↦    [1]:
↦       _[1]=0
↦       _[2]=s
↦       _[3]=0
↦    [2]:
↦       1
↦    [3]:
↦       1
↦ [3]:
↦    [1]:
↦       _[1]=s3-s
↦       _[2]=0
↦       _[3]=s2-1
↦    [2]:
↦       1
↦    [3]:
↦       1
```

See also: Section D.4.6.1 [normal], page 407; Section D.8.3.1 [parametrize], page 581; Section D.4.7.2 [primdecGTZ], page 410.

D.8.3.3 parametrizesing

Procedure from library `paramet.lib` (see Section D.8.3 [paramet_lib], page 580).

Usage: parametrizesing(f); f a polynomial in two variables, ordering ls or ds

Create: If the parametrization is successful, the basering will be changed to the parametrization ring, that is to the ring 0,(x,y),ls;

Return: a list containing the parametrizations of the different branches of the singularity at the origin resp. 0, if f was not of the desired kind

Example:

```
LIB "paramet.lib";
ring RING=0,(x,y),ls;
poly f=(x^2-y^3)*(x^2-y^2-y^3);
parametrizesing(f);
↦ HNE of one branch found
↦ HNE of one branch found
```

```
↦ finite HNE of one branch found
↦ // result: 3 branch(es) successfully computed,
↦ //         basering has changed to HNEring
↦ [1]:
↦    _[1]=x
↦    _[2]=x-1/2x2
↦ [2]:
↦    _[1]=x
↦    _[2]=-x-1/2x2
↦ [3]:
↦    _[1]=x3
↦    _[2]=x2
```

See also: Section D.5.5 [hnoether_lib], page 454; Section D.5.5.3 [reddevelop], page 457.

D.8.4 surf_lib

Library: surf.lib

Purpose: Procedures for Graphics with Surf

Author: Hans Schoenemann,
the program surf is written by Stefan Endrass

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

Procedures:

D.8.4.1 plot

Procedure from library `surf.lib` (see Section D.8.4 [surf_lib], page 583).

Usage: plot(I); I ideal or poly

Assume: I defines a plane curve or a surface given by one equation

Return: nothing

Note: requires the external program 'surf' to be installed

Example:

```
LIB "surf.lib";
// ---------  plane curves ------------
ring rr0 = 0,(x1,x2),dp;
ideal I = x1^3 - x2^2;
plot(I);
ring rr1 = 0,(x,y,z),dp;
ideal I(1) = 2x2-1/2x3 +1-y+1;
plot(I(1));
//  ---- Singular Logo --------------
poly logo = ((x+3)^3 + 2*(x+3)^2 - y^2)*(x^3 - y^2)*((x-3)^3-2*(x-3)^2-y^2);
plot(logo);
// Steiner surface
ideal J(2) = x^2*y^2+x^2*z^2+y^2*z^2-17*x*y*z;
```

```
plot(J(2));
// --------------------
plot(x*(x2-y2)+z2);
// E7
plot(x^3-x*y^3+z^2);
// Whitney umbrella
plot(z^2-x^2*y);
```

D.9 Coding theory

D.9.1 brnoeth_lib

Library: brnoeth.lib

Purpose: Brill-Noether Algorithm, Weierstrass-SG and AG-codes

Authors: Jose Ignacio Farran Martin, ignfar@eis.uva.es
Christoph Lossen, lossen@mathematik.uni-kl.de

Overview: Implementation of the Brill-Noether algorithm for solving the Riemann-Roch problem and applications in Algebraic Geometry codes. The computation of Weierstrass semigroups is also implemented.
The procedures are intended only for plane (singular) curves defined over a prime field of positive characteristic.
For more information about the library see the end of the file brnoeth.lib.

Main procedures: **Auxiliary procedures:** **See also:** Section D.5.5 [hnoether_lib], page 454; Section D.7.3 [triang_lib], page 559.

D.9.1.1 Adj_div

Procedure from library `brnoeth.lib` (see Section D.9.1 [brnoeth_lib], page 584).

Usage: Adj_div(f [,l]); f a poly, [l a list]

Return: list L with the computed data:
L[1] a list of rings: L[1][1]=aff_r (affine), L[1][2]=Proj_R (projective),
L[2] an intvec with 2 entries (degree, genus),
L[3] a list of intvec (closed places),
L[4] an intvec (conductor),
L[5] a list of lists:
L[5][d][1] a (local) ring over an extension of degree d,
L[5][d][2] an intvec (degrees of base points of places of degree d)

Note: `Adj_div(f);` computes and stores the fundamental data of the plane curve defined by f as needed for AG codes.
In the affine ring you can find the following data:
poly CHI: affine equation of the curve,
ideal Aff_SLocus: affine singular locus (std),
list Inf_Points: points at infinity
Inf_Points[1]: singular points
Inf_Points[2]: non-singular points,
list Aff_SPoints: affine singular points (if not empty).

In the projective ring you can find the projective equation CHI of the curve (poly).
In the local rings L[5][d][1] you find:

list POINTS: base points of the places of degree d,
list LOC_EQS: local equations of the curve at the base points,
list BRANCHES: Hamburger-Noether developments of the places,
list PARAMETRIZATIONS: local parametrizations of the places,

Each entry of the list L[3] corresponds to one closed place (i.e., a place and all its conjugates) which is represented by an intvec of size two, the first entry is the degree of the place (in particular, it tells the local ring where to find the data describing one representative of the closed place), and the second one is the position of those data in the lists POINTS, etc., inside this local ring.
In the intvec L[4] (conductor) the i-th entry corresponds to the i-th entry in the list of places L[3].

With no optional arguments, the conductor is computed by local invariants of the singularities; otherwise it is computed by the Dedekind formula.
An affine point is represented by a list P where P[1] is std of a prime ideal and P[2] is an intvec containing the position of the places above P in the list of closed places L[3].
If the point is at infinity, P[1] is a homogeneous irreducible polynomial in two variables.

If `printlevel>=0` additional comments are displayed (default: `printlevel=0`).

Example:

```
LIB "brnoeth.lib";
int plevel=printlevel;
printlevel=-1;
ring s=2,(x,y),lp;
list C=Adj_div(y9+y8+xy6+x2y3+y2+x3);
↦ The genus of the curve is 3
def aff_R=C[1][1];      // the affine ring
setring aff_R;
listvar(aff_R);         // data in the affine ring
↦ // aff_R                [0]  *ring
↦ // Inf_Points           [0]  list, size: 2
↦ // Aff_SPoints          [0]  list, size: 3
↦ // Aff_SLocus           [0]  ideal (SB), 2 generator(s)
↦ // CHI                  [0]  poly
CHI;                    // affine equation of the curve
↦ x3+x2y3+xy6+y9+y8+y2
Aff_SLocus;             // ideal of the affine singular locus
↦ Aff_SLocus[1]=y8+y2
↦ Aff_SLocus[2]=x2+y6
Aff_SPoints[1];         // 1st affine singular point: (1:1:1), no.1
↦ [1]:
↦    _[1]=y+1
↦    _[2]=x+1
↦ [2]:
↦    1
Inf_Points[1];          // singular point(s) at infinity: (1:0:0), no.4
↦ [1]:
↦    [1]:
```

```
↦       y
↦    [2]:
↦       4
Inf_Points[2];          // list of non-singular points at infinity
↦ empty list
//
def proj_R=C[1][2];     // the projective ring
setring proj_R;
CHI;                    // projective equation of the curve
↦ x3z6+x2y3z4+xy6z2+y9+y8z+y2z7
C[2][1];                // degree of the curve
↦ 9
C[2][2];                // genus of the curve
↦ 3
C[3];                   // list of computed places
↦ [1]:
↦    1,1
↦ [2]:
↦    1,2
↦ [3]:
↦    2,1
↦ [4]:
↦    1,3
C[4];                   // adjunction divisor (all points are singular!)
↦ 2,2,2,42
//
// we look at the place(s) of degree 2 by changing to the ring
C[5][2][1];
↦ //   characteristic : 2
↦ //   1 parameter    : a
↦ //   minpoly        : ...
↦ //   number of vars : 3
↦ //        block   1 : ordering ls
↦ //                  : names    x y t
↦ //        block   2 : ordering C
def S(2)=C[5][2][1];
setring S(2);
POINTS;                // base point(s) of place(s) of degree 2: (1:a:1)
↦ [1]:
↦    [1]:
↦       1
↦    [2]:
↦       (a)
↦    [3]:
↦       1
LOC_EQS;               // local equation(s)
↦ [1]:
↦    y2+y3+(a+1)*y4+y6+(a+1)*y8+y9+(a)*xy2+(a+1)*xy4+xy6+(a+1)*x2y+(a)*x2y2\
   +x2y3+x3
PARAMETRIZATIONS;      // parametrization(s) and exactness
↦ [1]:
↦    [1]:
↦       _[1]=t2+(a+1)*t3
```

```
↦       _[2]=t3+(a+1)*t4
↦    [2]:
↦       3,4
BRANCHES;              // Hamburger-Noether development
↦ [1]:
↦    [1]:
↦       _[1,1]=0
↦       _[1,2]=x
↦       _[1,3]=0
↦       _[2,1]=0
↦       _[2,2]=1
↦       _[2,3]=(a+1)
↦    [2]:
↦       1,-4
↦    [3]:
↦       0
↦    [4]:
↦       y+(a+1)*xy+(a)*x2y+(a)*x2y2+(a+1)*x3+x3y+x3y3+(a)*x4+(a+1)*x4y2+(a+\
   1)*x4y3+x5+x5y2+(a)*x6+(a+1)*x6y2+x6y4+x6y5+x7y+(a+1)*x8+(a+1)*x8y+x8y4+(\
   a+1)*x8y6+x9+x9y7+(a+1)*x10+x11y6+(a+1)*x12y4+x13y5+x14+x14y+x15y4+x16+(a\
   +1)*x16y2+x17y3+x19y2+(a+1)*x20+x21y+x23
printlevel=plevel;
```

See also: Section D.9.1.2 [NSplaces], page 587; Section D.9.1.10 [closed_points], page 596.

D.9.1.2 NSplaces

Procedure from library `brnoeth.lib` (see Section D.9.1 [brnoeth_lib], page 584).

Usage: NSplaces(h, CURVE), where h is an intvec and CURVE is a list

Return: list L with updated data of CURVE after computing all non-singular affine closed places whose degrees are in the intvec h:

in L[1][1]: (affine ring) lists Aff_Points(d) with affine non-singular (closed) points of degree d (if non-empty),
in L[3]: the newly computed closed places are added,
in L[5]: local rings created/updated to store (repres. of) new places.

See Section D.9.1.1 [Adj_div], page 584 for a description of the entries in L.

Note: The list_expression should be the output of the procedure Adj_div.
If `printlevel>=0` additional comments are displayed (default: `printlevel=0`).

Example:

```
LIB "brnoeth.lib";
int plevel=printlevel;
printlevel=-1;
ring s=2,(x,y),lp;
list C=Adj_div(x3y+y3+x);
↦ The genus of the curve is 3
// The list of computed places:
C[3];
↦ [1]:
↦    1,1
```

```
↦ [2]:
↦    1,2
// create places up to degree 4
list L=NSplaces(1..4,C);
// The list of computed places is now:
L[3];
↦ [1]:
↦    1,1
↦ [2]:
↦    1,2
↦ [3]:
↦    1,3
↦ [4]:
↦    2,1
↦ [5]:
↦    3,1
↦ [6]:
↦    3,2
↦ [7]:
↦    3,3
↦ [8]:
↦    3,4
↦ [9]:
↦    3,5
↦ [10]:
↦    3,6
↦ [11]:
↦    3,7
↦ [12]:
↦    4,1
↦ [13]:
↦    4,2
↦ [14]:
↦    4,3
// e.g., affine non-singular points of degree 4 :
def aff_r=L[1][1];
setring aff_r;
Aff_Points(4);
↦ [1]:
↦    [1]:
↦       _[1]=y2+y+1
↦       _[2]=x2+xy+x+1
↦    [2]:
↦       12
↦ [2]:
↦    [1]:
↦       _[1]=y4+y3+1
↦       _[2]=x+y3+y
↦    [2]:
↦       13
↦ [3]:
↦    [1]:
↦       _[1]=y4+y3+y2+y+1
```

```
↦       _[2]=x+y2+y+1
↦    [2]:
↦       14
// e.g., base point of the 1st place of degree 4 :
def S(4)=L[5][4][1];
setring S(4);
POINTS[1];
↦ [1]:
↦    (a3+a2+a+1)
↦ [2]:
↦    (a2+a)
↦ [3]:
↦    1
printlevel=plevel;
```

See also: Section D.9.1.1 [Adj_div], page 584; Section D.9.1.10 [closed_points], page 596.

D.9.1.3 BrillNoether

Procedure from library `brnoeth.lib` (see Section D.9.1 [brnoeth_lib], page 584).

Usage: BrillNoether(G,CURVE); G an intvec, CURVE a list

Return: list of ideals (each of them with two homogeneous generators, which represent the numerator, resp. denominator, of a rational function).
The corresponding rational functions form a vector basis of the linear system L(G), G a rational divisor over a non-singular curve.

Note: The procedure must be called from the ring CURVE[1][2], where CURVE is the output of the procedure `NSplaces`.
The intvec G represents a rational divisor supported on the closed places of CURVE[3] (e.g. `G=2,0,-1;` means 2 times the closed place 1 minus 1 times the closed place 3).

Example:

```
LIB "brnoeth.lib";
int plevel=printlevel;
printlevel=-1;
ring s=2,(x,y),lp;
list C=Adj_div(x3y+y3+x);
↦ The genus of the curve is 3
C=NSplaces(1..4,C);
// the first 3 Places in C[3] are of degree 1.
// we define the rational divisor G = 4*C[3][1]+4*C[3][3] (of degree 8):
intvec G=4,0,4;
def R=C[1][2];
setring R;
list LG=BrillNoether(G,C);
↦ Vector basis successfully computed
// here is the vector basis of L(G):
LG;
↦ [1]:
↦    _[1]=1
↦    _[2]=1
↦ [2]:
↦    _[1]=y
```

```
↦    _[2]=x
↦ [3]:
↦    _[1]=z
↦    _[2]=x
↦ [4]:
↦    _[1]=y2
↦    _[2]=x2
↦ [5]:
↦    _[1]=xz2+y3
↦    _[2]=x3
↦ [6]:
↦    _[1]=xyz2+y4
↦    _[2]=x4
printlevel=plevel;
```

See also: Section D.9.1.1 [Adj_div], page 584; Section D.9.1.2 [NSplaces], page 587; Section D.9.1.4 [Weierstrass], page 590.

D.9.1.4 Weierstrass

Procedure from library `brnoeth.lib` (see Section D.9.1 [brnoeth_lib], page 584).

Usage: Weierstrass(i, m, CURVE); i,m integers and CURVE a list

Return: list WS of two lists:
  WS[1] list of integers (Weierstr. semigroup of the curve at place i up to m)
  WS[2] list of ideals (the associated rational functions)

Note: The procedure must be called from the ring CURVE[1][2], where CURVE is the output of the procedure `NSplaces`.
i represents the place CURVE[3][i].
Rational functions are represented by numerator/denominator in form of ideals with two homogeneous generators.

Warning: The place must be rational, i.e., necessarily CURVE[3][i][1]=1.

Example:

```
LIB "brnoeth.lib";
int plevel=printlevel;
printlevel=-1;
ring s=2,(x,y),lp;
list C=Adj_div(x3y+y3+x);
↦ The genus of the curve is 3
C=NSplaces(1..4,C);
def R=C[1][2];
setring R;
// Place C[3][1] has degree 1 (i.e it is rational);
list WS=Weierstrass(1,7,C);
↦ Vector basis successfully computed
// the first part of the list is the Weierstrass semigroup up to 7 :
WS[1];
↦ [1]:
↦    0
```

```
↦ [2]:
↦    3
↦ [3]:
↦    5
↦ [4]:
↦    6
↦ [5]:
↦    7
// and the second part are the corresponding functions :
WS[2];
↦ [1]:
↦    _[1]=1
↦    _[2]=1
↦ [2]:
↦    _[1]=y
↦    _[2]=z
↦ [3]:
↦    _[1]=xy
↦    _[2]=z2
↦ [4]:
↦    _[1]=y2
↦    _[2]=z2
↦ [5]:
↦    _[1]=y3
↦    _[2]=xz2
printlevel=plevel;
```

See also: Section D.9.1.1 [Adj_div], page 584; Section D.9.1.3 [BrillNoether], page 589; Section D.9.1.2 [NSplaces], page 587.

D.9.1.5 extcurve

Procedure from library `brnoeth.lib` (see Section D.9.1 [brnoeth_lib], page 584).

Usage: extcurve(d, CURVE); d an integer, CURVE a list

Return: list L which is the update of the list CURVE with additional entries

L[1][3]: ring (p,a),(x,y),lp (affine),
L[1][4]: ring (p,a),(x,y,z),lp (projective),
L[1][5]: ring (p,a),(x,y,t),ls (local),
L[2][3]: int (the number of rational places),

the rings being defined over a field extension of degree d.
If d<2 then `extcurve(d,CURVE);` creates a list L which is the update of the list CURVE with additional entries

L[1][5]: ring p,(x,y,t),ls,
L[2][3]: int (the number of computed places over the base field).

In both cases, in the ring L[1][5] lists with the data for all the computed rational places (after a field extension of degree d) are created (see Section D.9.1.1 [Adj_div], page 584):

lists POINTS, LOC_EQS, BRANCHES, PARAMETRIZATIONS.

Note: The list CURVE should be the output of `NSplaces`, and must contain (at least) one place of degree d.
You actually need all the places with degree dividing d. Otherwise, not all the places are computed, but only part of them.
This procedure must be executed before constructing AG codes, even if no extension is needed. The ring L[1][4] must be active when constructing codes over the field extension.

Example:

```
LIB "brnoeth.lib";
int plevel=printlevel;
printlevel=-1;
ring s=2,(x,y),lp;
list C=Adj_div(x5+y2+y);
↦ The genus of the curve is 2
C=NSplaces(1..4,C);
// since we have all points up to degree 4, we can extend the curve
// to that extension, in order to get rational points over F_16;
C=extcurve(4,C);
↦ Total number of rational places : NrRatPl = 33
// e.g., display the basepoint of place no. 32:
def R=C[1][5];
setring R;
POINTS[32];
↦ [1]:
↦    (a3+a2+1)
↦ [2]:
↦    (a+1)
↦ [3]:
↦    1
printlevel=plevel;
```

See also: Section D.9.1.6 [AGcode_L], page 592; Section D.9.1.7 [AGcode_Omega], page 593; Section D.9.1.1 [Adj_div], page 584; Section D.9.1.2 [NSplaces], page 587; Section D.9.1.10 [closed_points], page 596.

D.9.1.6 AGcode_L

Procedure from library `brnoeth.lib` (see Section D.9.1 [brnoeth_lib], page 584).

Usage: AGcode_L(G, D, EC); G,D intvec, EC a list

Return: a generator matrix for the evaluation AG code defined by the divisors G and D.

Note: The procedure must be called within the ring EC[1][4], where EC is the output of `extcurve(d)` (or within the ring EC[1][2] if d=1).
The entry i in the intvec D refers to the i-th rational place in EC[1][5] (i.e., to POINTS[i], etc., see Section D.9.1.5 [extcurve], page 591).
The intvec G represents a rational divisor (see Section D.9.1.3 [BrillNoether], page 589 for more details).
The code evaluates the vector basis of L(G) at the rational places given by D.

Warnings: G should satisfy $2 * genus - 2 < deg(G) < size(D)$, which is not checked by the algorithm.
G and D should have disjoint supports (checked by the algorithm).

Example:

```
LIB "brnoeth.lib";
int plevel=printlevel;
printlevel=-1;
ring s=2,(x,y),lp;
list HC=Adj_div(x3+y2+y);
↦ The genus of the curve is 1
HC=NSplaces(1..2,HC);
HC=extcurve(2,HC);
↦ Total number of rational places : NrRatPl = 9
def ER=HC[1][4];
setring ER;
intvec G=5;      // the rational divisor G = 5*HC[3][1]
intvec D=2..9;   // D = sum of the rational places no. 2..9 over F_4
// let us construct the corresponding evaluation AG code :
matrix C=AGcode_L(G,D,HC);
↦ Vector basis successfully computed
// here is a linear code of type [8,5,>=3] over F_4
print(C);
↦ 0,0,(a+1),(a),  1,  1,    (a),  (a+1),
↦ 1,0,(a),  (a+1),(a),(a+1),(a),  (a+1),
↦ 1,1,1,    1,    1,  1,    1,    1,
↦ 0,0,(a),  (a+1),1,  1,    (a+1),(a),
↦ 0,0,1,    1,    (a),(a+1),(a+1),(a)
printlevel=plevel;
```

See also: Section D.9.1.7 [AGcode_Omega], page 593; Section D.9.1.1 [Adj_div], page 584; Section D.9.1.3 [BrillNoether], page 589; Section D.9.1.5 [extcurve], page 591.

D.9.1.7 AGcode_Omega

Procedure from library `brnoeth.lib` (see Section D.9.1 [brnoeth_lib], page 584).

Usage: AGcode_Omega(G, D, EC); G,D intvec, EC a list

Return: a generator matrix for the residual AG code defined by the divisors G and D.

Note: The procedure must be called within the ring EC[1][4], where EC is the output of `extcurve(d)` (or within the ring EC[1][2] if d=1).
The entry i in the intvec D refers to the i-th rational place in EC[1][5] (i.e., to POINTS[i], etc., see Section D.9.1.5 [extcurve], page 591).
The intvec G represents a rational divisor (see Section D.9.1.3 [BrillNoether], page 589 for more details).
The code computes the residues of a vector space basis of
$\Omega(G-D)$ at the rational places given by D.

Warnings: G should satisfy $2*genus-2 < deg(G) < size(D)$, which is not checked by the algorithm.
G and D should have disjoint supports (checked by the algorithm).

Example:

```
LIB "brnoeth.lib";
int plevel=printlevel;
printlevel=-1;
ring s=2,(x,y),lp;
```

```
list HC=Adj_div(x3+y2+y);
↦ The genus of the curve is 1
HC=NSplaces(1..2,HC);
HC=extcurve(2,HC);
↦ Total number of rational places : NrRatPl = 9
def ER=HC[1][4];
setring ER;
intvec G=5;      // the rational divisor G = 5*HC[3][1]
intvec D=2..9;   // D = sum of the rational places no. 2..9 over F_4
// let us construct the corresponding residual AG code :
matrix C=AGcode_Omega(G,D,HC);
↦ Vector basis successfully computed
// here is a linear code of type [8,3,>=5] over F_4
print(C);
↦ 1,    0,  1,    1,  (a+1),(a),0,  0,
↦ (a+1),1,  (a),  0,  (a),  0,  (a),0,
↦ 0,    (a),(a+1),(a),(a),  0,  0,  1
printlevel=plevel;
```

See also: Section D.9.1.6 [AGcode_L], page 592; Section D.9.1.1 [Adj_div], page 584; Section D.9.1.3 [BrillNoether], page 589; Section D.9.1.5 [extcurve], page 591.

D.9.1.8 prepSV

Procedure from library `brnoeth.lib` (see Section D.9.1 [brnoeth_lib], page 584).

Usage: prepSV(G, D, F, EC); G,D,F intvecs and EC a list

Return: list E of size n+3, where n=size(D). All its entries but E[n+3] are matrices:

E[1]: parity check matrix for the current AG code

E[2] ... E[n+2]: matrices used in the procedure decodeSV

E[n+3]: intvec with

E[n+3][1]: correction capacity

epsilon

of the algorithm

E[n+3][2]: designed Goppa distance

delta

of the current AG code

Note: Computes the preprocessing for the basic (Skorobogatov-Vladut) decoding algorithm. The procedure must be called within the ring EC[1][4], where EC is the output of `extcurve(d)` (or in the ring EC[1][2] if d=1)
The intvec G and F represent rational divisors (see Section D.9.1.3 [BrillNoether], page 589 for more details).
The intvec D refers to rational places (see Section D.9.1.7 [AGcode_Omega], page 593 for more details.). The current AG code is `AGcode_Omega(G,D,EC)`.
If you know the exact minimum distance d and you want to use it in `decodeSV` instead of *delta* , you can change the value of E[n+3][2] to d before applying decodeSV.
If you have a systematic encoding for the current code and want to keep it during the

decoding, you must previously permute D (using `permute_L(D,P);`), e.g., according to the permutation P=L[3], L being the output of `sys_code`.

Warnings: F must be a divisor with support disjoint from the support of D and with degree $epsilon + genus$, where

$epsilon := [(deg(G) - 3 * genus + 1)/2]$.
G should satisfy $2 * genus - 2 < deg(G) < size(D)$, which is not checked by the algorithm.
G and D should also have disjoint supports (checked by the algorithm).

Example:

```
LIB "brnoeth.lib";
int plevel=printlevel;
printlevel=-1;
ring s=2,(x,y),lp;
list HC=Adj_div(x3+y2+y);
↦ The genus of the curve is 1
HC=NSplaces(1..2,HC);
HC=extcurve(2,HC);
↦ Total number of rational places : NrRatPl = 9
def ER=HC[1][4];
setring ER;
intvec G=5;      // the rational divisor G = 5*HC[3][1]
intvec D=2..9;   // D = sum of the rational places no. 2..9 over F_4
// construct the corresp. residual AG code of type [8,3,>=5] over F_4:
matrix C=AGcode_Omega(G,D,HC);
↦ Vector basis successfully computed
// we can correct 1 error and the genus is 1, thus F must have degree 2
// and support disjoint from that of D;
intvec F=2;
list SV=prepSV(G,D,F,HC);
↦ Vector basis successfully computed
↦ Vector basis successfully computed
↦ Vector basis successfully computed
// now everything is prepared to decode with the basic algorithm;
// for example, here is a parity check matrix to compute the syndrome :
print(SV[1]);
↦ 0,0,(a+1),(a),  1,  1,    (a),  (a+1),
↦ 1,0,(a),  (a+1),(a),(a+1),(a),  (a+1),
↦ 1,1,1,    1,    1,  1,    1,    1,
↦ 0,0,(a),  (a+1),1,  1,    (a+1),(a),
↦ 0,0,1,    1,    (a),(a+1),(a+1),(a)
// and here you have the correction capacity of the algorithm :
int epsilon=SV[size(D)+3][1];
epsilon;
↦ 1
printlevel=plevel;
```

See also: Section D.9.1.7 [AGcode_Omega], page 593; Section D.9.1.9 [decodeSV], page 596; Section D.9.1.5 [extcurve], page 591; Section D.9.1.13 [permute_L], page 599; Section D.9.1.12 [sys_code], page 598.

D.9.1.9 decodeSV

Procedure from library `brnoeth.lib` (see Section D.9.1 [brnoeth_lib], page 584).

Usage: decodeSV(y, K); y a row-matrix and K a list

Return: a codeword (row-matrix) if possible, resp. the 0-matrix (of size 1) if decoding is impossible.
For decoding the basic (Skorobogatov-Vladut) decoding algorithm is applied.

Note: The list_expression should be the output K of the procedure `prepSV`.
The matrix_expression should be a (1 x n)-matrix, where n = ncols(K[1]).
The decoding may fail if the number of errors is greater than the correction capacity of the algorithm.

Example:

```
LIB "brnoeth.lib";
int plevel=printlevel;
printlevel=-1;
ring s=2,(x,y),lp;
list HC=Adj_div(x3+y2+y);
↦ The genus of the curve is 1
HC=NSplaces(1..2,HC);
HC=extcurve(2,HC);
↦ Total number of rational places : NrRatPl = 9
def ER=HC[1][4];
setring ER;
intvec G=5;      // the rational divisor G = 5*HC[3][1]
intvec D=2..9;   // D = sum of the rational places no. 2..9 over F_4
// construct the corresp. residual AG code of type [8,3,>=5] over F_4:
matrix C=AGcode_Omega(G,D,HC);
↦ Vector basis successfully computed
// we can correct 1 error and the genus is 1, thus F must have degree 2
// and support disjoint from that of D
intvec F=2;
list SV=prepSV(G,D,F,HC);
↦ Vector basis successfully computed
↦ Vector basis successfully computed
↦ Vector basis successfully computed
// now we produce 1 error on the zero-codeword :
matrix y[1][8];
y[1,3]=a;
// and then we decode :
print(decodeSV(y,SV));
↦ 0,0,0,0,0,0,0,0
printlevel=plevel;
```

See also: Section D.9.1.7 [AGcode_Omega], page 593; Section D.9.1.5 [extcurve], page 591; Section D.9.1.8 [prepSV], page 594.

D.9.1.10 closed_points

Procedure from library `brnoeth.lib` (see Section D.9.1 [brnoeth_lib], page 584).

Usage: closed_points(I); I an ideal

Return: list of prime ideals (each a Groebner basis), corresponding to the (distinct affine closed) points of V(I)

Note: The ideal must have dimension 0, the basering must have 2 variables, the ordering must be lp, and the base field must be finite and prime.
It might be convenient to set the option(redSB) in advance.

Example:

```
LIB "brnoeth.lib";
ring s=2,(x,y),lp;
// this is just the affine plane over F_4 :
ideal I=x4+x,y4+y;
list L=closed_points(I);
// and here you have all the points :
L;
↦ [1]:
↦    _[1]=y+1
↦    _[2]=x+1
↦ [2]:
↦    _[1]=y+1
↦    _[2]=x
↦ [3]:
↦    _[1]=y+1
↦    _[2]=x2+x+1
↦ [4]:
↦    _[1]=y
↦    _[2]=x+1
↦ [5]:
↦    _[1]=y
↦    _[2]=x
↦ [6]:
↦    _[1]=y
↦    _[2]=x2+x+1
↦ [7]:
↦    _[1]=y2+y+1
↦    _[2]=x
↦ [8]:
↦    _[1]=y2+y+1
↦    _[2]=x+1
↦ [9]:
↦    _[1]=y2+y+1
↦    _[2]=x+y+1
↦ [10]:
↦    _[1]=y2+y+1
↦    _[2]=x+y
```

See also: Section D.7.3 [triang_lib], page 559.

D.9.1.11 dual_code

Procedure from library `brnoeth.lib` (see Section D.9.1 [brnoeth_lib], page 584).

Usage: dual_code(G); G a matrix of numbers

Return: a generator matrix of the dual code generated by G

Note: The input should be a matrix G of numbers.
The output is also a parity check matrix for the code defined by G

Example:

```
LIB "brnoeth.lib";
ring s=2,T,lp;
// here is the Hamming code of length 7 and dimension 3
matrix G[3][7]=1,0,1,0,1,0,1,0,1,1,0,0,1,1,0,0,0,1,1,1,1;
print(G);
↦ 1,0,1,0,1,0,1,
↦ 0,1,1,0,0,1,1,
↦ 0,0,0,1,1,1,1
matrix H=dual_code(G);
print(H);
↦ 1,1,1,0,0,0,0,
↦ 1,0,0,1,1,0,0,
↦ 0,1,0,1,0,1,0,
↦ 1,1,0,1,0,0,1
```

D.9.1.12 sys_code

Procedure from library `brnoeth.lib` (see Section D.9.1 [brnoeth_lib], page 584).

Usage: sys_code(C); C is a matrix of constants

Return: list L with:
L[1] is the generator matrix in standard form of an equivalent code,
L[2] is the parity check matrix in standard form of such code,
L[3] is an intvec which represents the needed permutation.

Note: Computes a systematic code which is equivalent to the given one.
The input should be a matrix of numbers.
The output has to be interpreted as follows: if the input was the generator matrix of an AG code then one should apply the permutation L[3] to the divisor D of rational points by means of `permute_L(D,L[3]);` before continuing to work with the code (for instance, if you want to use the systematic encoding together with a decoding algorithm).

Example:

```
LIB "brnoeth.lib";
ring s=3,T,lp;
matrix C[2][5]=0,1,0,1,1,0,1,0,0,1;
print(C);
↦ 0,1,0,1,1,
↦ 0,1,0,0,1
list L=sys_code(C);
L[3];
↦ 2,4,3,1,5
// here is the generator matrix in standard form
print(L[1]);
↦ 1,0,0,0,1,
↦ 0,1,0,0,0
// here is the control matrix in standard form
```

```
print(L[2]);
↦ 0, 0,1,0,0,
↦ 0, 0,0,1,0,
↦ -1,0,0,0,1
// we can check that both codes are dual to each other
print(L[1]*transpose(L[2]));
↦ 0,0,0,
↦ 0,0,0
```

See also: Section D.9.1.7 [AGcode_Omega], page 593; Section D.9.1.13 [permute_L], page 599; Section D.9.1.8 [prepSV], page 594.

D.9.1.13 permute_L

Procedure from library `brnoeth.lib` (see Section D.9.1 [brnoeth_lib], page 584).

Usage: permute_L(L, P); L,P either intvecs or lists

Return: list obtained from L by applying the permutation given by P.

Note: If P is a list, all entries must be integers.

Example:

```
LIB "brnoeth.lib";
list L=list();
L[1]="a";
L[2]="b";
L[3]="c";
L[4]="d";
intvec P=1,3,4,2;
// the list L is permuted according to P :
permute_L(L,P);
↦ [1]:
↦    a
↦ [2]:
↦    c
↦ [3]:
↦    d
↦ [4]:
↦    b
```

See also: Section D.9.1.7 [AGcode_Omega], page 593; Section D.9.1.8 [prepSV], page 594; Section D.9.1.12 [sys_code], page 598.

7 Release Notes

7.1 News and changes

NEWS in SINGULAR 2-0-3

The current version 2-0-3 is an update to the new release series version 2-0.

Singular version 2 is generally much faster than any previous version of Singular, due to a rewrite of major parts of the Singular kernel (e.g., the kernel has a new memory manager and a new polynomial arithmetic) and due to new and better implemented algorithms (e.g., for computing resolutions and determinants).

Besides theses internal changes, Singular version 2 offers many new features and functionalities (which were partly already incorporated in the 1-3 series). E.g., a native Windows distribution, an Emacs user interface, a new help system, 16 new libraries, etc.

All these changes are explained in more detail below.

Efficiency Improvements

The following table shows some timings which compare the efficiency of Singular version 1-2-3 and Singular version 2. All times are in seconds and were obtained on an AMD Athlon with 700 MHz, 128 MB RAM, running Linux.

Example	**Comp**	**Ring**	**1-2**	**2-0**	**Speedup**
mora_1	std	32003,x(1..3),ds	82	9.4	8.7
mora_9	std	32003,x(1..14),ds	9600	96	100
homog_gonnet	std	32003,x(1..18),dp	33	8.9	4.5
homog_cyclic_8	std	32003,x(1..9),dp	1873	472	4.0
cyclic_8	std	32003,x(1..8),dp	>47800	478	>100

General Changes

Windows distribution
: created with InstallShield, includes Cygwin, Singular, and (optionally) XEmacs, distributed as several self-extracting archives. Contains SINGULAR help in a Windows Help format (the file `Manual.hlp`).

Section 3.2 [Emacs user interface], page 21
: the recommended interface for using SINGULAR

`ESingular`
: new program for an out-of-the-box, pre-customized Emacs which runs Singular

Section 3.1.3 [The online help system], page 14
: choose in which browser the on-line help is displayed;
wildcard expansion of help topics
new layout of html manual pages

Section 3.10.2 [Source code debugger], page 64
: interactive debugging of procedures written in the SINGULAR language

Section 5.1.25 [example], page 138
    Provide on-line examples for all kernel and library commands.

file/directory layout of distribution
    `Singular/<version>/`

http://www.singular.uni-kl.de
    New WWW home-site of SINGULAR

RPM versions of Singular
    including integration into KDE

Extensions of the ring concept

new coefficient domains: arbitrary long real and complex numbers

no restriction on number of ring variables

no restriction on number of parameters

New SINGULAR libraries

Reorganization of libraries into "Categories".

Section D.4.10 [intprog_lib], page 422 procedures for computing toric ideals

Section D.4.11 [toric_lib], page 424 procedures for Integer Programming using Groebner bases

Section D.5.10 [spectrum_lib], page 495 procedures for computing spectra

Section D.4.5 [mregular_lib], page 404 procedures for computing the Castelnuovo-Mumford regularity

Section D.5.6 [mondromy_lib], page 473
    procedures for computing the monodromy of a singularity

Section D.5.4 [gaussman_lib], page 440
    procedures for the Gauss-Manin connection of a singularity

Section D.5.9 [spcurve_lib], page 490
    procedures for CM codimension 2 singularities

Section D.7.3 [triang_lib], page 559
    procedures for decomposing zero-dimensional ideals

Section D.7.2 [solve_lib], page 546
    procedures for solving polynomial systems

Section D.8.4 [surf_lib], page 583
    visualization of curves and surfaces with `surf`

Section D.4.9 [reesclos_lib], page 418
    Rees Algebra and integral closure of an ideal

Section D.9.1 [brnoeth_lib], page 584
    Brill-Noether algorithm, Weierstrass semigroups and AG codes

Section D.6.4 [stratify_lib], page 529
    Algorithmic stratification by the Greuel-Pfister algorithm

Section D.8.3 [paramet_lib], page 580
    parametrization of curves

Section D.6.3 [rinvar_lib], page 523
Invariant rings of reductive groups

Section D.7.5 [zeroset_lib], page 562
Procedures for roots and factorization

Section D.5.7 [qhmoduli_lib], page 476
Moduli spaces of sqh-singularities

Changes in SINGULAR libraries

Many procedures were moved into different libraries, the documentation of libraries was generally improved and some libraries renamed:

Section D.6.2 [ainvar_lib], page 518
renamed `invar.lib` to `ainvar.lib` (see also `rinvar.lib`, `finvar.lib`).

Section D.5.2 [deform_lib], page 434
renamed procedures `T1, T2, T12` to `T_1, T_2, T_12`.

Section D.2.4 [poly_lib], page 311
new procedures numerator/denominator

Section D.3.2 [linalg_lib], page 353
included `jordan.lib` in `linalg.lib`

New SINGULAR functions

`convhull, simplex`
TODO: docu, tests

Section 5.1.20 [division], page 134
Extension of lift: `division(M,SM)` returns a list `[T,U,R]` with `SM*U+R=M*T`

Section 5.2.2 [breakpoint], page 215
sets a debugger breakpoint

`div` integer division omitting remainder

Section 5.1.24 [ERROR], page 137
interrupts computation in current procedure and returns with error message to top-level

Section 5.1.33 [fglmquot], page 142
calculate ideal quotients using FGLM-techniques

Section 5.1.64 [leadmonom], page 162
returns the leading monomial of a polynomial or a vector as a polynomial or vector whose coefficient is one

Section 5.1.44 [highcorner], page 149
returns the smallest monomial not contained in and ideal/module

Section 5.1.47 [hres], page 151
calculate free resolution of homogeneous ideal using Hilbert-driven algorithm.

Section 5.1.73 [minor], page 168
with an optional third argument minors modulo a standard basis will be computed

Section 5.1.77 [mpresmat], page 170
calculate module representing the multipolynomial resultant matrix

Section 5.1.130 [uressolve], page 210
compute all complex roots of a zerodimensional ideal

Section 5.1.131 [vandermonde], page 210
solve Vandermonde linear system

Section 5.1.55 [jet], page 156
extended for series expansions of rational expressions

Section 5.1.106 [reduce], page 191
extended for reduced normal form of rational expressions

Singular functions whose syntax/semantics has changed

Section 5.1.20 [division], page 134
arguments swapped w.r.t. version 2-0-1 and before

Section 5.1.10 [coeffs], page 128
Extended possible syntax to `coeffs(ideal,ideal)` and `coeffs(module,module)` which is equivalent to `coeffs(ideal,ideal,p)` and `coeffs(module,module,p)` where `p` is a polynomial (i.e., monomial) consisting of all ring variables.

Section 5.1.26 [execute], page 138
parenthesis around argument of execute is now **mandatory**.

Section 5.1.66 [lift], page 163
new implementation (unified handling of isSB and not isSB case); new optional third argument (a matrix name), which, if given, stores the "matrix of units"

Section 5.1.2 [bareiss], page 122
new implementation after Lie-Saunders, much faster

Section 5.1.25 [example], page 138
run example for all kernel commands.

Section 5.1.97 [print], page 183
Finer-grained control for formatted output.

Section D.2.3.10 [pause], page 310
can be called with argument, must be called with parenthesis.

Section 5.1.89 [option], page 177
new options: `oldStd`, `redThrough`, `notBuckets`

Changes of command-line options and system resources

`--browser`, `--allow-net`, `--emacs`, `--sdb`
new command-line options (see Section 3.1.6 [Command line options], page 18) .

Section 5.1.124 [system], page 205
values of command-line options can be inspected/set at run-time with `system(<option>[,<new_value])`.

`.singularrc`
system administrators can put a "global" `.singularrc` file into the directories where Singular libraries reside (see Section 3.1.7 [Startup sequence], page 21) .

`libreadline`
: if found, a shared version of the `readline` library is linked-in dynamically at run-time. If not found, `readline` is emulated. This feature is not available on Windows 95/98/ME/NT/2K/XP and MacOS X.

Internal Changes

new data structures for monomials and polynomials
: Makes polynomial arithmetic significantly faster: Especially for block - or weighted orderings. Bucket representation of polynomials in std; Ring change during std, res and bareiss computations (resulting in more speed and less space consumption).

new memory management
: Results is less memory usage, faster memory allocation/free, less fragmentation, much better locality of reference.

factorization
: revisited, missing cases implemented

Porting

- Singular is available for ix86-libc5, ix86-libc6, HPUX_9, HPUX_10, SunOS-4, SunOS-5, IRIX-6, ix86-Win (Windows 95/98/ME/NT/2K/XP), FreeBSD 4.2, MacOS X
- `dbm` functionality is not available on Windows 95/98/ME/NT/2K/XP and MacOS X
- `MP` functionality is not available on Windows 95/98/ME/NT/2K/XP and MacOS X.

7.2 Downloading instructions

Singular is available as binary program for most common hard- and software platforms. Release versions of Singular can be downloaded with ftp from our FTP site

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

or, using your favorite WWW browser, from

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

To download Singular for a Unix platform

Make sure that you have approximately 20 MByte of free disk space and follow these steps.

1. You need to download two (archive) files:

   `Singular-2-0-3-share.tar.gz`
   : contains architecture independent data like documentation and libraries
   `ftp://www.mathematik.uni-kl.de/pub/Math/Singular/UNIX/Singular-2-0-3-share.tar.gz`

   `Singular-2-0-3-<uname>.tar.gz`
   : contains architecture dependent executables, like the Singular program.

   `<uname>` is a description of the processor and operating system for which Singular is compiled. Choose one of the following:

   `ix86-Linux`
   : PC's running under Linux with `libc` version 6 (e.g., for SuSe Linux distribution version >= 6; RedHat Linux distribution version >= 5.2).

     `ftp://www.mathematik.uni-kl.de/pub/Math/Singular/Singular-2-0-3-ix86-Linux.tar.gz`

`ix86-Linux-libc5` PC's running under Linux with `libc` version 5 (if you are not sure which `libc` version you have, run `ldd /bin/ls`).
`ftp://www.mathematik.uni-kl.de/pub/Math/Singular/UNIX/Singular-2-0-3-ix86-Linux-libc5.tar.gz`

`ix86-freebsd` 's running under FreeBSD
`ftp://www.mathematik.uni-kl.de/pub/Math/Singular/UNIX/Singular-2-0-3-ix86-freebsd.tar.gz`

`ix86-Win` PC's running Windows 95/98/ME/NT/2K/XP which have Cygwin version 1.3.9 (or higher) already installed. Unless you are familiar with Cygwin, we recommend that you download one of the self-extracting archives as described below.

`HPUX-9` HP workstations running under HPUX version 9
`ftp://www.mathematik.uni-kl.de/pub/Math/Singular/UNIX/Singular-2-0-3-HPUX-9.tar.gz`

`HPUX-10` HP workstations running under HPUX version 10
`ftp://www.mathematik.uni-kl.de/pub/Math/Singular/UNIX/Singular-2-0-3-HPUX-10.tar.gz`

`SunOS-5` Sun workstations running Solaris version 5
`ftp://www.mathematik.uni-kl.de/pub/Math/Singular/UNIX/Singular-2-0-3-SunOS-5.tar.gz`

`IRIX-6` IRIX workstations running IRIX version 6
`ftp://www.mathematik.uni-kl.de/pub/Math/Singular/UNIX/Singular-2-0-3-IRIX-6.tar.gz`

`AIX-4` AIX workstations running AIX version 4
`ftp://www.mathematik.uni-kl.de/pub/Math/Singular/UNIX/Singular-2-0-3-AIX-4.tar.gz`

`PowerMacintosh-darwin` PowerMacintosh running Mac OS X
`ftp://www.mathematik.uni-kl.de/pub/Math/Singular/MAC/OsX/Singular-2-0-3-PowerMacintosh-darwin.tar.gz`

`DecAlpha-Linux` Digital Dec Alpha running Linux (RedHat 5.2)
`ftp://www.mathematik.uni-kl.de/pub/Math/Singular/UNIX/Singular-2-0-3-DecAlpha-Linux.tar.gz`

`DecAlpha-OSF1` Digital Dec Alpha running OSF 1
`ftp://www.mathematik.uni-kl.de/pub/Math/Singular/UNIX/Singular-2-0-3-DecAlpha-OSF1.tar.gz`

Please contact us if you can not find an appropriate architecture dependent archive.

2. Simply change to the directory in which you wish to install Singular (usually wherever you install 3rd-party software):

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

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

3. Unpack the archives:

   ```
   gzip -dc Singular-2-0-3-<uname>.tar.gz | tar -pxf -
   gzip -dc Singular-2-0-3-share.tar.gz | tar -pxf -
   ```

   If you have downloaded `Singular-2-0-3-share.zip`, unpack with

   ```
   unzip -u Singular-2-0-3-share.zip
   ```

4. After unpacking, see the created file `Singular/2-0-3/INSTALL` (which is also located at `ftp://www.mathematik.uni-kl.de/pub/Math/Singular/INSTALL_unix.html` for details on how to finish the installation.

To download Singular for Windows 95/98/ME/NT/2K/XP

1. Download one of the following self-extracting archives:

   `Singular-2-0-3-Compact.exe`
   `(ftp://www.mathematik.uni-kl.de/pub/Math/Singular/WINDOWS/Singular-2-0-3-Compact.exe)`
   Minimal archive to download. Installs Singular and minimal set of needed tools and DLLs.

`Singular-2-0-3-Typical.exe`
(`ftp://www.mathematik.uni-kl.de/pub/Math/Singular/WINDOWS/Singular-2-0-3-Typical.exe`)
Typical archive to download. Installs Singular and the XEmacs editor to run ESingular.

2. Double-click (or, execute), the self-extracting archives, and **carefully** follow the instructions given there.
3. In case of problems, see the created file `/etc/INSTALL.txt` (which is also located at `ftp://www.mathematik.uni-kl.de/pub/Math/Singular/WINDOWS/INSTALL_win.html` for details on how to finish the installation.

To download SINGULAR for the Macintosh

For the SINGULAR version for Mac OS X refer to the Unix section.
The SINGULAR version for Mac OS 9 is available as `ftp://www.mathematik.uni-kl.de/pub/Math/Singular/MAC/classic/Singular-2-0-2-html.sea.hqx` (documentation) and `ftp://www.mathematik.uni-kl.de/pub/Math/Singular/MAC/classic/Singular-2-0-2-ppc-MPW.sea.hqx` (binaries and libraries). The file `ftp://www.mathematik.uni-kl.de/pub/Math/Singular/MAC/classic/INSTALL_mac.html` gives details on how to finish the installation.

7.3 Unix installation instructions

To install SINGULAR on a Unix platform, you need the following two archives:

`Singular-2-0-3-share.tar.gz`
contains architecture independent data like documentation and libraries (alternatively, you may also download `Singular-2-0-3-share.zip`).

`Singular-2-0-3-<uname>.tar.gz`
contains architecture dependent executables, like the SINGULAR program.

`<uname>` is a description of the processor and operating system for which SINGULAR is compiled (e.g, `ix86-Linux`). Please contact us if you can not find an appropriate architecture dependent archive.

You can obtain these (and other) archives from

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

To install SINGULAR

Make sure that you have approximately 12MByte of free disk space and follow these steps.

1. Simply change to the directory in which you wish to install SINGULAR (usually wherever you install 3rd-party software), for example:

```
cd /usr/local
or
mkdir install;cd install
(you do not need root privileges in this case)
```

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

2. Unpack the archives:

```
gzip -dc <path_to>/Singular-2-0-3-<uname>.tar.gz | tar -pxf -
gzip -dc <path_to>/Singular-2-0-3-share.tar.gz | tar -pxf -
```

This creates the the directory Singular/2-0-3 with

(sub)directories	which contain
`<uname>`	`Singular` and `ESingular` programs
`LIB`	SINGULAR libraries (`*.lib` files)
`emacs`	files for the SINGULAR Emacs user interface
`info`	info files of SINGULAR manual
`html`	html files of SINGULAR manual
`doc`	miscellaneous documentation files
`examples`	SINGULAR examples (`*.sing` files)

For the executable to work, the directory layout must look pretty much like this; the executable looks for "sibling" directories at run-time to figure out where its SINGULAR libraries and on-line documentation files are. These constraints on the local directory layout are necessary to avoid having to hard-code pathnames into the executables, or require that environment variables be set before running the executable. In particular, you **must not move or copy** the SINGULAR executables to another place, but use soft-links instead.

The following steps are optional:

- Arrange that typing `Singular` at the shell prompt starts up the installed SINGULAR executable.

  If you have root permission, do:

  ```
  ln -s `pwd`/Singular/2-0-3/<uname>/Singular  /usr/local/bin/Singular-2-0-3
  ln -s `pwd`/Singular/2-0-3/<uname>/ESingular  /usr/local/bin/ESingular-2-0-3
  ln -s /usr/local/bin/Singular-2-0-3 /usr/local/bin/Singular
  ln -s /usr/local/bin/ESingular-2-0-3 /usr/local/bin/ESingular
  ```

  Otherwise, append the directory `` `pwd`/Singular/2-0-3/<uname>/ `` to your `$PATH` environment variable. For the `csh` (or, `tcsh`) shell do:

  ```
  set path=(`pwd`/Singular/2-0-3/<uname> $path)
  ```

  For the `bash` (or, `ksh`) shell do:

  ```
  export PATH=`pwd`/Singular/2-0-3/<uname>/:$PATH
  ```

  You also might want to adjust your personal start-up files (`~/.cshrc` for `csh`, `~/.tcshrc` for `tcsh`, or `~/.profile` for `bash`) accordingly, so that the `$PATH` variable is set automatically each time you login.

  IMPORTANT: Do *never* move or copy the file `Singular/2-0-3/<uname>/Singular` to another place, but use soft-links instead.

- If you wish to use any of following features of SINGULAR, make sure that the respective programs are installed on your system:

Feature	**Requires**
running `ESingular`, or `Singular` within Emacs	Emacs version 20 or higher, or, Xemacs version 20.3 or higher
on-line `html` help	netscape browser
on-line `info` help	info, or tkinfo texinfo browser programs
TAB completion and history mechanism of ASCII-terminal interface	shared readline library, i.e. `/usr/lib/libreadline.so`
visualization of curves and surfaces	surf version 0.9 or higher (only available for Linux and Solaris).

  You may download most of these programs from

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

- Customize the on-line help system:

  By default, on-line help is displayed in the `html` format using the `netscape` program.

  However, this behavior can be customized in several ways using the Singular commands `system("--browser",<browser>)` and `system("--allow-net", 1)` (or, by starting up Singular with the respective command line options).

  In particular, creating the file `Singular/2-0-3/LIB/.singularrc` and putting the Singular command

  ```
  system("--allow-net", 1);
  ```

  in it, allows the on-line help system to fetch its `html` pages from Singular's WWW home site (`http://www.singular.uni-kl.de`) in case its local html pages are not found. That is, you may delete your local html pages, after setting this option.

  See also Section 3.1.3 [The online help system], page 14, Section 3.1.6 [Command line options], page 18, and Section 3.1.7 [Startup sequence], page 21, for more details on customizing the on-line help system.

- Add the line

  ```
  * Singular:(singular.hlp).     A system for polynomial computations
  ```

  to your system-wide `dir` file (usually `/usr/info/dir` or `/usr/local/info/dir` and copy or soft-link the file `Singular/2-0-3/info/singular.hlp` to the directory of your `dir` file.
  This assures that the Singular manual can be accessed from stand-alone texinfo browser programs such as `info` or `Emacs`. (It is not necessary for the use of the help system from within `Singular`.)

Troubleshooting

- General: Singular can not find its libraries or on-line help
  1. Make sure that you have read and/or execute permission the files and directories of the Singular distribution. If in doubt, `cd` to the directory where you unpacked Singular, and do (as root, if necessary):

     ```
     chmod -R a+rX Singular
     ```

  2. Start up Singular, and issue the command `system("Singular");`. If this does not return the correct and expanded location of the Singular executable, then you found a bug in Singular, which we ask you to report (see below).
  3. Check whether the directories containing the libraries and on-line help files can be found by Singular: If `$bindir` denotes the directory where the Singular executable resides, then Singular looks for library files as follows:
     (0) the current directory
     (1) all dirs of the environment variable SINGULARPATH
     (2) `$bindir/LIB`
     (3) `$bindir/../LIB`
     (4) `/usr/local/Singular/2-0-3/LIB`
     (4) `/usr/local/Singular/LIB`
     The on-line `info` files need to be at `$bindir/../info` and the `html` pages at `$bindir/../html`.

  You can inspect the found library and `info/html` directories by starting up Singular with the `--version` option, or by issuing the Singular command `system("--version");`.

- In case you have troubles running ESingular:

1. Check if you have XEmacs version 20.3 (or newer) or Emacs version 20 (or newer) installed on your system. The latest versions are available from `http://www.xemacs.org`, resp. `http://www.fsf.org/software/emacs/emacs.html`.
2. ESingular uses a number of standard Lisp packages, which are provided with most XEmacs and Emacs distributions but for some reasons might be not present on your computer. For your convenience the archives `edit-utils-1.82-pkg.tar.gz` and `os-utils-1.26-pkg.tar.gz` containing the necessary libraries have been included in the present distribution of SINGULAR-2-0-3 in the folder `UNIX/emacs` but they can also be downloaded from `ftp://ftp.xemacs.org/packages`. You need then to extract the archives into a directory found by your XEmacs (resp. Emacs) `load-path`.

   ```
   cd <path_to_xemacs/emacs_libraries>
   gzip -dc <path_to>/edit-utils-1.82-pkg.tar.gz.tar | tar -pxf -
   gzip -dc <path_to>/os-utils-1.26-pkg.tar.gz | tar -pxf -
   ```

   If you are in doubt where your XEmacs (resp. Emacs) libraries reside, try to localize files with ending `.elc` on your system.

- Under SuSE-Linux, ESINGULAR does not display a prompt: This is due to the very restrictive access rights of `/dev/pty*` of the standard SuSE (`http:/www.suse.de`) distribution (starting from version 6.3 on). As root, do one of the following: Either

  ```
  chmod 666 /dev/pty*
  ```

  or,

  ```
  chmod g+s $(which emacs)
  chgrp tty $(which emacs)
  chmod g+s $(which xemacs)
  chgrp tty $(which xemacs)
  ```

  See also `http://sdb.suse.de/sdb/de/html/ke_devpts-63.html` for details.
- For `HPUX` systems:
  The /etc/termcap files which are pre-installed on these systems are broken which leads to messages like

  ```
  Name collision between 9826 9836
  ```

  when SINGULAR is started. To fix this, you can get fixed termcap files from `ftp://www.mathematik.uni-kl.de/pub/Math/Singular/utils` by downloading the files termcap-<uname-description> and either replace your /etc/termcap file by the file found there, or view, and redo the respective changes by `diff`'ing the downloaded and the original file.
- For `ix86-Linux` systems:
  Due to some incompatibilities of shared libraries, the start-up of SINGULAR might fail with messages like

  ```
  Can not find shared library ...
  ```

  For DEBIAN systems, try to do `ln -s /usr/lib/libncurses.so /usr/lib/libncurses.so.4`.
  If this fails (and on other systems) download and install `Singular-2-0-3-ix86-Linux-static.tar.gz`.
- For `AIX` systems:
  The default `info` program of the system is not GNU's texinfo browser which is used to display the on-line documentation in the `info` format. Therefore, the distribution of the AIX executable already contains the `info` browser program. If you remove this program, make sure that the GNU `info` program is executed if you call '`info`' from your shell.

- For any other troubles:
  Please send an email to `singular@mathematik.uni-kl.de` and include the header which is displayed by starting up SINGULAR with the `-v` option, and a description of your machine (issue the command `uname -a` on your shell) in your report.

7.4 Windows installation instructions

For the impatient:

If you run Windows NT/2K/XP, make sure you are logged in with the `Administrator` rights.

If you have some older version of `Singular` installed, please uninstall it before installing `Singular-2-0-3`. It is recommended to install the new `Singular` in the same directory where the previous version was installed. Thus your existing environment will be used automatically.

If you have `Cygwin` or `XEmacs` installed and they do not belong of some previous `Singular` installation, please read the information below.

Execute the self-extracting installation archive and **carefully** follow the instructions given there. Setup will analyze your system, create the corresponding configuration and give you further tips on the installation process.

In case of troubles, see the installed file `/etc/INSTALL.txt` or visit the **Singular Forum** at `http://www.singular.uni-kl.de/forum`.

Installation preliminaries

Running `Singular` on MS Windows requires that the Cygwin DLL (dynamically linked library) version 1.3.9 or higher is installed on your system (see `http://sources.redhat.com/cygwin/` for details about Cygwin). We install and configure `Cygwin` version 1.3.9 automatically, if the `BasicCygwin` component is selected.

To run `ESingular` (i.e. `Singular` within its Emacs interface) on MS Windows requires furthermore that the `XEmacs` editor is installed on your system. We install and configure `XEmacs` version 21.1.13 automatically, if the `XEmacs` component is selected.

The `Singular` installation is divided into the following components (you see them explicitly if you choose the Custom Setup Type during the installation):

Program Files:

`Singular`
: Contains the `Singular` Program, documentation and example files.
  All files of this component are installed under the directory `/usr/local/Singular/2-0-3`

`BasicCygwin`
: Contains the Cygwin DLL and basic tools/programs (like `bash` or `info`) of the Cygwin environment.
  The Cygwin DLL and programs are installed in the directory `/bin`. Miscellaneous documentation and configuration files are installed into the directory `/etc`.

`XEmacs`
: Contains the Cygwin port of the `XEmacs` editor (see `http://www.xemacs.org` for more about `XEmacs`) version 21.1.13.
  All files of this component are installed under the directory `/usr/local`.

`ESingular`
: Contains the `ESingular`, the most comfortable Singular interface. Requires `XEmacs` to run.

Remark: All essential DLLs, some programs and symbolic links are installed in the directory `/usr/local/Singular/2-0-3/ix86-Win`, which must be present in your `PATH` system variable.

Help Files:

`WinHelp` Contains Singular Manual, converted to Windows help format (.hlp). We recommend to install it, because it is small in size. Moreover, we set it as default help browser in all the Singular executables.

`HTML Help` Contains the "native" Singular Manual. It's not included by default, because it is much bigger than WinHelp, though it has better typesetting of mathematical formulae.

The Typical Setup Type of the Singular-2-0-3-Typical.exe contains `BasicCygwin`, `Singular`, `XEmacs`, `ESingular`, `WinHelp` and `HTML Help` components. The Compact Setup Type does not contain only the `HTML Help` component.

Downloading preliminaries

To install `Singular` on a PC running Windows 95/98/ME/NT/2K/XP, you need to download **one** of the following archive files:

Minimal archive: Singular-2-0-3-Compact.exe (`Singular-2-0-3-Compact.exe`) (`BasicCygwin`, `Singular`, `WinHelp` components)

Typical archive : Singular-2-0-3-Typical.exe (`Singular-2-0-3-Typical.exe`) (`BasicCygwin`, `Singular`, `XEmacs`, `ESingular`, `WinHelp`, `HTML Help` components)

You can obtain these (and other) archives either from the `WINDOWS` directory on the `Singular` CD or from

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

Installing Singular

The recent Setup program analyzes your system, e.g. it detects whether Cygwin and XEmacs are present and functional. Then it gives you hints and/or error messages with instructions during the installation.

`If you have Cygwin installed`
: If the version of `cygwin1.dll` is not less than 1.3.9, you will be prompted that the `BasicCygwin` component will not be installed. Moreover, the existing Cygwin root directory will be proposed as the default installation directory. You can accept it, then `Singular` will be placed into the existing Cygwin environment. If you choose another destination, `Singular` will use Cygwin programs anyway. If your Cygwin has incorrect settings (like inconsistent mount points among the important ones), `Singular` Setup will correct them.

`If you have XEmacs installed`
: The `Cygwin` port of `XEmacs` version 21.1.13 together with the most important packages is included in the distribution. The use of non-`Cygwin` ports or earlier `Cygwin`

ports has some quirks and does not work properly together with `Singular` (i.e., running `ESingular` results in problems). Nevertheless, if you want to test the compatibility of `ESingular` with your own `XEmacs`, just de-select the XEmacs component, but leave `ESingular` selected. If your `XEmacs` happens to be incompatible with ESingular, you can install just the XEmacs component and use in fact two different `XEmacs` programs. The setup program will search for `XEmacs` and packages in `<TARGETDIR>/usr/local/lib` (here `<TARGETDIR>` is the directory which you selected while running setup). If you have `Cygwin` and the Cygwin port of `XEmacs`, you might have `XEmacs` installed there. To avoid overwriting your `XEmacs` files, setup will ask you what to do.

Customizing Singular

The following step(s) are optional:

To conveniently access all files on your computer, mount the drives of your disk(s):

1. Open a rxvt terminal (from Start->Programs->Singular 2.0->Tools).
2. For each drive (letter) issue the following command (the drive `C:` is used in the example):

```
mount -f -s -t C: /C
```

After mounting the drive(s) this way, you can conveniently access the files on a drive from the `bash` shell and from `XEmacs` using the `/C/` prefix. Otherwise, you need to type `/cygdrive/C` - that is the Cygwin convention.

Create your personal `HOME` directory.
Various applications (e.g., `bash` or `XEmacs`) store user-specific configuration/customization files in the `HOME` directory of the current user. During the installation, the `HOME` directory is globally set for all users to `/home/Singular`. To create your individual `HOME` directory follow these steps:

1. Create a directory (folder), say, `C:\home\MyHome`.
2. Copy all files from `/home/Singular` to `C:\home\MyHome`.
3. Set the *user* environment variable `HOME` to `C:\home\MyHome`:
   On Windows 95/98/ME, edit the respective line the `AUTOEXEC.BAT` file.
   On Windows NT/2K/XP add an entry in your `Environment` control panel.
4. Restart your computer (On Windows 95/98/ME).

You may furthermore examine the files copied into your `HOME` directory and adjust (i.e. edit) them to your personal preferences/needs. System-wide settings are stored in the file `/etc/profile`.

Uninstalling Singular

You can uninstall individual or all installed components by opening the `Software` shortcut of the `Control panel`. Simply double-click the `Singular` item and follow the instructions given there.

For any other troubles, please send an email to `singular@mathematik.uni-kl.de` and include the header which is displayed by starting up `Singular` with the `-v` option, and a description of your machine and operating system.

7.5 Macintosh installation instructions

For the SINGULAR version for Mac OS X refer to the Unix section. The SINGULAR version for Mac OS 9 is available as 'ftp://www.mathematik.uni-kl.de/pub/Math/Singular/MAC/classic/Singular-2-0-3-html.sea.hqx' (documentation) and 'ftp://www.mathematik.uni-kl.de/pub/Math/Singular/MAC/classic/Singular-2-0-3-ppc-MPW.sea.hqx' (binaries and libraries). The file 'ftp://www.mathematik.uni-kl.de/pub/Math/Singular/MAC/classic/INSTALL_mac.txt' gives details on how to finish the installation.

Installation preliminaries

This is the final distribution of SINGULAR as MPW tool. Newer versions of SINGULAR will only be available for MacOS X (which is handled like other UNIX-systems). To install SINGULAR on a Macintosh (PPC only), you need the file `Singular-2-0-2-ppc-MPW.sea.hqx` resp. `Singular-2-0-2-ppc-MPW.hqx` which you can obtain by public ftp from `ftp://www.mathematik.uni-kl.de/pub/Math/Singular`. Furthermore, you need MacOS version >= 8.5 with an installed MPW shell. This version works also without Carbon.

Installing SINGULAR on MacOS X

Please see `INSTALL_unix.html`. In contrast to other Unix systems, under MacOS X, SINGULAR is only available as a shell command and there is no Emacs interface. Enter `Singular` in a terminal under Aqua to run SINGULAR. Alternatively, you may choose `Terminal` as the application to open the file `Singular` with and put it into the dock.

Installing Singular

1. Make sure that you have MPW installed on your system. See below for instructions on how to install MPW and how to avoid conflicts with the `Carbon.lib`.
2. Download the file `Singular-2-0-2-ppc-MPW.sea.hqx`.
3. Expand the downloaded file using a standard Expander (like Compact Pro or StuffIt) to obtain the corresponding self-extracting `Singular-2-0-2-ppc-MPW.sea` file.
4. Click on the self-extracting `Singular-2-0-2-ppc-MPW.sea` file. This will create a new folder named `Singular:` which contains the SINGULAR program in the subfolder `2-0-2:ppc-MPW:`. Do not change the structure of the subfolders.
5. Make the folder which contains the executable to the working directory of the MPW shell. To start SINGULAR, you need to type `Singular` in the MPW worksheet. Clicking on the SINGULAR icon on the desktop, starts up the MPW shell (the working directory of the MPW shell is then the directory of the executable). If you start the MPW shell manually, you must set the working directory to the directory of the SINGULAR program. You may use the directory menu or write into the startup file of the MPW shell the line

   ```
   setdirectory "{Boot}"Singular:2-0-2:ppc-MPW:
   ```

   if `"{Boot}"Singular:2-0-2:ppc-MPW:` is the folder of the executable.

   After unpacking, you will have a directory structure like:

   `Singular:2-0-2:ppc-MPW:Singular`
   : Singular executable

   `Singular:2-0-2:ppc-MPW:libparse`
   : parser for Singular libraries

`Singular:2-0-2:LIB:`
: Singular libraries

`Singular:2-0-2:LIB:gftables:`
: Singular gftables

`Singular:2-0-2:doc:singular.idx`
: index of inline help

`Singular:2-0-2:doc:singular.hlp`
: file for inline help

`Singular:2-0-2:README`
: information of the distribution

`Singular:2-0-2:README.hlml`

`Singular:2-0-2:COPYING`
: licence information

`Singular:2-0-2:COPYING.html`

`Singular:2-0-2:NEWS`
: news in the distribution

`Singular:2-0-2:NEWS.html`

`Singular:2-0-2:INSTALL`
: instructions like this file

The SINGULAR libraries have a special form to work here. Do not use the libraries from other distributions. They are not compatible. A supplementary file is `Singular-2-0-2-html.sea.hqx` which contains the html pages. You can use this with a browser but not from the program SINGULAR. The self-extracting archive will create the folder `html:` and you may extract it at `"{Boot}"Singular:2-0-2:`.

Installing MPW

The final version of MPW is downloadable from Apple at `ftp://ftp.apple.com/developer/Tool_Chest/Core_Mac_OS_Tools/MPW_etc./MPW-GM_Images/`. Older versions of `Carbon.lib` may conflict with the latest MPW. In this case, you need to update `Carbon.lib` or work without it.

Restrictions

Some typical Unix features of SINGULAR are not working with MPW, e.g. links and cooperation with other programs like XEmacs and xterm.

For any other troubles, please send an email to `singular@mathematik.uni-kl.de` and include the header which is displayed by starting up SINGULAR with the `-v` option, and a description of your machine and operating system.

8 Index

|

~

"

+

>

^

\

<

A

B

C

D

E

F

G

H

I

J

N

O

P

T

U

V

W

X

Z

Table of Contents