Singular - Tutorial

Stefan Steidel
Department of Mathematics
University of Kaiserslautern
67653 Kaiserslautern
steidel@mathematik.uni-kl.de

SINGULAR version 3-0-4
University of Kaiserslautern
Department of Mathematics and Centre for Computer Algebra
Authors: G.–M. Greuel, G. Pfister, H. Schönemann
Copyright ©1986-2007

October 20, 2008

Contents

1 Preface

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

If you want to be informed of new releases, please register 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).

For the citation of Singular see

`http://www.singular.uni-kl.de/how_to_cite.html`,

for information on how to cite Singular.

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

1.1 Overview of Singular

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

Here are some of the most important features of Singular:

- Main computational objects: ideals/modules over very general polynomial rings over various ground fields.
- Large variety of algorithms implemented in kernel (written in C/C++).
- Many more algorithms implemented as Singular libraries.
- Intuitive, C-like programming language.
- Extensive documentation: Manual (info, ps, and html), Publications.
- Available for most hard- and software platforms: Unix (HP-UX, SunOS, Solaris, Linux, AIX), Windows, Macintosh.

Singular is a free software under the GNU General Public Licence.

Singular is developed by the Singular team at the Department of Mathematics of the University of Kaiserslautern under the direction of Gert–Martin Greuel, Gerhard Pfister and Hans Schönemann.

1.2 Availability

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

1.3 History of Singular

1983 Greuel/Pfister: Existence of complete intersection singularities which are not quasi-homogeneous, but Poincare-complex exact?

1983 Neuendorf (geb. Schemmel)/Pfister: Implementation of the Gröbner basis–algorithm BuchMora for ZX–Spectrum in Basic.

1987 Pfister et al (Humboldt Uni Berlin): Buchmora for Atari in Modula-2: Existence shown.

1989 Buchmora renamed to Singular; Developed jointly by groups from Berlin (Pfister) and Kaiserslautern (Greuel).

1990 Ported to Unix; First user manual.

1993 Rewritten in C; Singular programming language – Libraries; Faster than Macaulay.

1996 Multivariate polynomial factorization; gcd.

1997 Release of Singular version 1.0 (multivariate polynomial factorization; gcd, syzygies, resolutions, communication links).

1998 Release of Singular version 1.2 (faster, primary decomposition, normalization).

1999 Release of Singular version 1.4 (much faster, numerical data types and algorithms, monodromy, moduli of space curves, debugger).

2001 Release of Singular version 2.0.

2002 Book *A* Singular *Introduction to Commutative Algebra* (by G.-M. Greuel and G. Pfister, with contributions by O. Bachmann, C. Lossen and H. Schönemann). The book includes a CD containing a distribution of Singular version 2-0-3.

2004 The Richard D. Jenks Prize for Excellence in Software Engineering for Computer Algebra was awarded to the Singular team.

2005 Singular 3-0-0 for Mac OS X is available in the unstable branch of fink.

2005 Singular is included in the Knoppix/Math CD.

2005 The Singular web pages have been revised and equipped with a new css-based design.

2007 Release of Singular version 3-0-3: available for most Unix platforms, Windows and Mac OS X.

2007 Release of Singular version 3-0-4: available for most Unix platforms, Windows and Mac OS X.

1.4 Acknowledgement

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

Currently, the Singular team has the following members: Michael Brickenstein, Wolfram Decker, Alexander Dreyer, Anne Frühbis-Krüger, Kai Krüger, Viktor Levandovskyy, Oleksandr Motsak, Mathias Schulze and Oliver Wienand.

Former members of the Singular team are: Olaf Bachmann, Christoph Lossen, Wolfgang Neumann, Wilfred Pohl, Jens Schmidt, Thomas Siebert, Rüdiger Stobbe, Eric Westenberger and Tim Wichmann.

Further contributions to Singular were made by: Thomas Bayer, Isabelle Bermejo, Markus Becker, Stas Bulygin, Kai Dehmann, Marcin Dumnicki, Stephan Endrass, Jose Ignacio Farran, Vladimir Gerdt, Philippe Gimenez, Christian Gorzel, Hubert Grassmann, Fernando Hernando, Agnes Heydtmann, Dietmar Hillebrand, Tobias Hirsch, Manuel Kauers, Simon King, Anen Lakhal, Martin Lamm, Santiago Laplagne, Gregoire Lecerf, Francisco Javier Lobillo, Thomas Markwig, Bernd Martin, Michael Messollen, Andrea Mindnich, Jorge Martin Morales, Thomas Nüssler, Carlos Rabelo, Alfredo Sanchez-Navarro, Henrik Strohmayer, Christian Stussak, Imade Sulandra, Christine Theis, Enrique Tobis, Alberto Vigneron-Tenorio, Moritz Wenk, Denis Yanovich, Oleksandr Iena.

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 Getting started

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

2.1 First steps

2.1.1 Notations and basic concepts

- Singular input and output as well as set words will be written in typewriter face, e.g. `exit;` or `help;`.
- The whole Singular manual [GPS1] is available online by typing the command `help;` or `?;`. Explanation on single topics, e.g. on `intmat`, which defines a matrix of integers, are obtained by

  ```
  help intmat;
  ```

  This shows the text from node `intmat`, in the printed manual. A more detailed treatment of the command `help;` will be done in section 2.1.3.
- The symbol `//->` starts Singular output, e.g.:

  ```
  int i=5;
  i;
  //-> 5
  ```

- Square brackets are access operators for strings, integer vectors, ideals, matrices, polynomials, resolutions, and lists, e.g.:

  ```
  intvec v = 3,5,6,7,8;
  v[4];
  //-> 7
  ```

- Keys are also shown in typewriter face, such as:

  `n` (press the key `n`),
  `RETURN` (press the enter key),
  `CTRL-P` (press the control key and `P` simultaneously).
- 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 respectively 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.

2.1.2 Starting and terminating Singular

Obviously, the first question is, how does one start the programme and how is it terminated? The most current version of Singular is started by using the command

```
Singular
```

in the command line of the system. Entering

```
Singular -v
```

will prompt use of the version number of Singular when starting the programme.

After the start, Singular shows an input prompt, a >, back and is available to the user for interactive use. As soon as the user no longer wants to use this possibility, he is recommended to terminate the programme. There are three commands available for this: `exit;`, `quit;` or, for very lazy users, `$;`.

Please note that the semicolons in the preceding paragraph are part of the Singular commands, and not inserted punctuations marks.

In general, *every* command in Singular ends with a semicolon!

The semicolon tells the computer that the inputted command is to be *interpreted* and, if this is successful, be *carried out*. The programme comes up with a result (possibly an error notification) followed by a new input prompt. Should the user forget the semicolon, or have opened with a curved bracket and not closed with one, SINGULAR shows this with an input prompt ., in words a dot, and enables further inputs, such as the missing semicolon. In this way it is possible to stretch longer commands over several lines.

2.1.3 The online help `help`

The next most important information after the start and terminate commands is how to find help if the user is stuck. Here SINGULAR offers the command `help`, or in short `?`. Using the command `help` followed by a SINGULAR command, a SINGULAR function or procedure name or a SINGULAR library, information to the respective objects are shown. For the libraries one receives a list of the procedures contained therein, for commands, functions and procedures one is shown their purpose and finds the general syntax as well as, very important, examples for their use.

Examples:

```
? exit;
? standard.lib;
? printf;
```

The help can be shown on various browsers. Standard is SINGULAR 3.0.4 Netscape. This means that SINGULAR, after inputting e.g. `help exit;`, starts Netscape and shows the help text to the `exit;`. (Via self–explanatory buttons the entire handbook is available.) Apart from Netscape, further browsers are available from which only `info` and `builtin` are named here. Users of Unix operating systems are probably familiar with the first, the last shows the help text on the current SINGULAR page and has the advantage of functioning on all computer platforms and without additional programmes (such as Netcape or Info).

Through the command `system("browsers");` one learns which browsers are known to SINGULAR, and through `system("--browser","builtin");` the browser can be changed from Netscape to `builtin` — in the same manner for other browsers. In addition, it is possible to define the browser when starting SINGULAR, by starting the programme with the following command

```
Singular --browser=builtin
```

Whilst use of the Netscape help is self–explanatory, pointers are necessary, if the user has never worked with Info. Anyone not working with Info may go immediately to 2.1.4. To move around within Info, the following commands should be used, which are all just one letter. Note that the `RETURN` or arrow keys should *never* be used! Some commands read subsequently further input from the command line at the bottom of the monitor. Here the `TAB` key can be used to complete a partially inputted command.

Some important Info commands:

`q`	leave the online help
`n`	leaf on to the next menu button
`p`	leaf back to preceding menu button
`m`	choose a specified menu button by name
`f`	call up a cross–reference
`l`	call up the last visited menu button
`b`	leaf back to the start of the menu buttons
`e`	leaf forward to the end of the menu buttons
`SPACE`	scroll forward one page
`DEL`	scroll back one page
`h`	call up the Info introduction
`CTRL-H`	call up a short synopsis on the online help
`s`	search the handbook for a defined string
`1,...,9`	call up the i-th subbutton of a menu

2.1.4 Interrupt Singular

Under Unix–like operating systems and under Windows NT, it is possible, via the key combination `CTRL-C`. to force an interruption in Singular. (Does not work wit `ESingular`!) Singular reacts with an output of the currently performed command and awaits further instructions. Following options are available:

`a`	Singular carries out the current command and returns then to top level,
`c`	Singular carries on,
`q`	the programme Singular is terminated.

2.1.5 Editting inputs

If a command has been miswritten, or if an earlier command is needed again, it is not absolutely necessary to renew the input. Existing Singular text can be edited. For this, Singular records a history of all commands of a Singular session. Below is a selection of the available key combinations for text editing:

`TAB`	automatic completion of function and file names
$\leftarrow$	
`CTRL-B`	moves the cursor to the left
$\rightarrow$	
`CTRL-F`	moves the cursor to the right
`CTRL-A`	moves the cursor to the beginning of the line
`CTRL-E`	moves the cursor to the end of the line
`CTRL-D`	deletes the letter under the cursor – never use in an empty line!
`BACKSPACE`	
`DEL`	
`CTRL-H`	deletes the letter in front of the cursor

`CTRL-K`	deletes all from the cursor to the end of the line
`CTRL-U`	deletes all from the cursor to the beginning of the line
↓	
`CTRL-N`	supplies the next line from the history
↑	
`CTRL-P`	supplies the preceding line from the history
`RETURN`	sends the current line to the SINGULAR–Parser

2.1.6 Types of data in Singular and rings

SINGULAR operates with a whole range of diverse structures, which are available as various data types. If an object in SINGULAR is to be defined, that is, a variable entered, it is necessary to allocate it to a data type right from the start.

Data types in SINGULAR depend on a meta structure, the so-called ring, over which they exist, with the exception of **`string`**, **`int`**, **`intvec`** und **`intmat`**. To perform a calculation in SINGULAR it is first absolutely necessary to define the ring over which one is working.

`ring r=0,x,lp;`	The amount of polynomials in the variables x with coefficients in the rational numbers $\mathbb{Q}$ and lexicographical ordering.
`ring r=(0,a,b),(x,y,z),lp;`	The amount of polynomials in the variables x, y, z, where the coefficients are rational terms in the variables a and b. Of course, instead of a, b resp. x, y, z, also any other variables can be shown. Important is that the variables in the first brackets can appear in the denominator of fractions, the ones in the second brackets may not.
`ring r=32003,x(1..10),dp;`	The amount of polynomials in the variables $x_1, \ldots, x_{10}$ with coefficients in the field of characteristic 32003 and degree reverse lexicographical ordering.
`ring r=(real,15),x,lp;`	The amount of polynomials in the variables x with coefficients in real numbers $\mathbb{R}$ — for calculations with 15 places after the decimal point.

We shall calculate first over the rational numbers $\mathbb{Q}$. We do not need real numbers as floating point numbers or even complex numbers until later.

A list of the available data types in SINGULAR is given below, and we also show for each an example, by defining a variable of the relative type and, through the operator `=`, allocating a value to it.

`int i=1;`	The data type `integer` represents the machine numbers (= whole numbers between -2^{31} und $2^{31}-1$). In addition, true values (= `boolean`) are represented as `integers`, $0 = \text{FALSE}$, `1 = TRUE`.
`string s="Hallo";`	`strings` are any chains of letters, always boxed in by inverted commas.
`intvec iv=1,2,3,4;`	A vector of `integers`.
`intmat im[2][3]=1,2,3,4,5,6;`	A matrix with two lines and three columns with `integer` entries, $\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix}$.
`ring R=(0,a),(x,y),lp;`	The ring $\mathbb{Q}(a)[x,y]$ with lexicographical order. For further explanations please see the handbook [GPS1].
`number n=4/6;`	`numbers` are the elements of the field based on the ring. By `ring r=0,x,lp;` the rational numbers, by `ring r=(0,a),x,lp;` also fractions of polynomials in a with complete number coefficients, i.e. $\frac{a^2+1}{a-1}$.
`list l=n,iv,s;`	A list can contain objects of different types. Through `l[2]`, the second entry of `l` can be seized.
`matrix m[2][3]=1,2,3,4,5,6;`	A matrix with two lines and three columns, the entries being either of type `poly` or of type `number`, as shown here $\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix}$.
`vector v=[1,2,3];`	A vector in module $\mathtt{R}^3$. If the entries are all of type `number`, we can interpret it as vector over the field.
`module mo=v,[x,x2,x+1];`	The mounted module in $\mathtt{R}^3$ by `v` and $(x, x^2, x+1)^t$.
`poly f=x2+2x+1;`	A polynomial in the indeterminates of the ring with `numbers` as coefficients, here $f = x^2 + 2x + 1$. Note that numbers in front of the monomials are interpreted as coefficients, whereas SINGULAR interprets numbers after single variables as exponents.

`ideal i=f,x3;`	That from f and x^3 generated ideal in `R`.
`qring Q=i;`	The quotient ring `R/i`.
`map g=R,x;`	The rest class formation of `R` after `Q`, which is defined by $x \mapsto \overline{x}$.
`def j;`	If, at the point time when the definition of a variable is fixed, one does not want to fix the type, it can be defined as `def`. The first allocation of a value to the variable also fixes the data type.
`link`	For the data type `link`, we refer to the handbook [GPS1].
`resolution`	For the data type `resolution`, we refer to the handbook [GPS1].

At first glance it might seem as though the matrices `im` and `m` are identical. In the case of SINGULAR that is not the case as they are of different types!

If one wishes to calculate with decimal points, having the ground field $\mathbb{R}$ available, then one must replace, in the definition of the ring, the characteristic 0 with `real` (resp. `(real,50)`, if one wishes to calculate with 50 decimal points), e.g.

```
ring r=(real,10),x,lp;
```

Even the complex numbers are available by replacing `real` with `complex`. `i` defines then the imaginary unit, i.e. the square root of -1.

2.1.7 Procedures

If there exists a definite problem which is to be solved with SINGULAR, then one calls up the programme, enters the command sequence and obtains a result. Very often, the same calculations are to be carried out with different inputs. Then it makes sense to write the command sequence as a procedure, into which the desired inputs are entered as arguments and which returns the solutions.

The syntax of a procedure is fairly simple:

```
proc PROCEDURENAME [PARAMETERLIST]
{
  PROCEDUREBODY
}
```

For `PROCEDURENAME`, any not otherwise delegated letter sequence can be used. The types and names of the arguments which are passed on to the procedure are laid down in the `PARAMETERLIST`. The `PARAMETERLIST` should be encased in round brackets. The `PROCEDUREBODY` contains a sequence of permissible SINGULAR code. If the procedure is to return a result, the result should be stored in a variable `result` and the procedure should terminate with the command `return(result);`.

An example is more useful than thousands of words:

```
proc permcol (matrix A, int c1, int c2)
{
  matrix B=A;
  B[1..nrows(B),c1]=A[1..nrows(A),c2];
  B[1..nrows(B),c2]=A[1..nrows(A),c1];
  return(B);
}
```

The procedure `permcol` should exchange two columns of a matrix. For this three arguments are necessary. The first argument is given the name `A` and is of the type `matrix`, the two following are `c1` and `c2` and are of type `integer`. Singular instructions follows and the result is stored in the variable `B` of type `matrix`, which is then returned with `return(B);`. This means, in particular, that the result of the procedure is of type `matrix` 2.1.6).

A procedure can be called up by entering the procedure name, followed by the arguments in round brackets. E.g.

```
LIB "matrix.lib";
LIB "inout.lib";
ring r=0,x,lp;
matrix A[3][3]=1,2,3,4,5,6,7,8,9;
pmat(A,2);
//->  1,  2,  3,
      4,  5,  6,
      7,  8,  9
matrix B=permcol(A,2,3);
pmat(B,2);
//->  1,  3,  2,
      4,  6,  5,
      7,  9,  8
```

Variables, which are defined within a procedure, are only known there and can, therefore, have the same name as objects which are defined outside the procedure.

Here are two more trivial examples.

(a) Trace of a matrix:

Our aim is to write a procedure that computes the trace of a quadratic matrix $M \in \mathrm{Mat}(n \times n, \mathbb{Z})$. Therefore we have to consider first how such a matrix is declared:

```
intmat M[3][3] = 1,2,3,4,5,6,7,8,9;
M;
//-> 1,2,3,
//-> 4,5,6,
//-> 7,8,9
```

Note that `intmat` is independent of a given basering.
A possible procedure that returns the trace of a quadratic input-matrix $A \in \mathrm{Mat}(n \times n, \mathbb{Z})$ is as follows:

```
proc trma(intmat A)
{
  if (nrows(A) != ncols(A))
  {
    print("Attention: A is not a square matrix");
    return(0);
  }
  int re = 0;
  int i = 1;
  while (i <= nrows(A))
  {
    re = re + A[i,i];
    i++;
  }
  return(re);
}
```

Hence, we obtain:

```
trma(M);
//-> 15
trace(M);   //the built-in SINGULAR command
//-> 15
```

(b) Greatest common divisor of two integers:

Our aim is to write a prodedure that computes the greatest common divisor of two integers $a, b \in \mathbb{Z}$. A possible realization is the following:

```
proc GCD(int a, int b)
{
  int r = a % b;
  while (r != 0)
  {
    a = b;
    b = r;
    r = a % b;
  }
  return(b);
}

GCD(15, 21);
//-> 3
GCD(2765851, 255781);
//-> 31
gcd(2765851, 255781);  //the built-in SINGULAR command
//-> 31
```

2.1.8 Libraries

To make procedures available for more than one Singular session, it makes sense to store them in files, which can be re-read by so-called Singular — Libraries. The library names often allow conclusions to be drawn from the procedures contained and always have the ending `.lib`. Libraries are read into Singular through the command `LIB` followed by the encapsulated library name in inverted commas `"`, such as

```
LIB "123456.lib";
```

(Library names should, if possible, only consist of *eight* letters, to guarantee compatibility with systems such as Dos!) If they are not Singular's own libraries, then they should be in the register from which Singular is started.

Of course, a library must conform to certain syntax rules, and procedures, which are stored in libraries, should be extended by two explanatory additions. We show this in an example:

```
/////////////////////////////////////////////////////////////////
version="1.0";

info="
    LIBRARY:  123456.lib
    AUTHOR:   Stefan Steidel, email: steidel@mathematik.uni-kl.de
    PROCEDURES:
      permcol(matrix,int,int)  interchanges columns of the matrix
      permrow(matrix,int,int)  interchanges rows of the matrix
";

/////////////////////////////////////////////////////////////////
LIB "inout.lib";
/////////////////////////////////////////////////////////////////

proc permcol (matrix A, int c1, int c2)
"USAGE:  permcol(A,c1,c2);  A matrix, c1,c2 positive integers
RETURN:  matrix,  A being modified by permuting column c1 and c2
NOTE:    space for important remarks
         also over more than one row
EXAMPLE: example permcol; shows an example
"
{
  matrix B=A;
  B[1..nrows(B),c1]=A[1..nrows(A),c2];
  B[1..nrows(B),c2]=A[1..nrows(A),c1];
  return(B);
}

example
{ "EXAMPLE:"; echo = 2;
   ring r=0,x,lp;
   matrix A[3][3]=1,2,3,4,5,6,7,8,9;
   pmat(A);
   pmat(permcol(A,2,3));
}
.
.
.
```

If a double slash // in a line appears, the rest of the line is interpreted as a comment and ignored.

The first section, to be found between both the comment lines, is the so–called head of the library. The first line contains the set word `version`, through which the version number of the library is fixed. General information to the library follows the set word `info`.

It should be noted that under the item `PROCEDURES:` all procedure names contained in the library are depicted with a maximum of one–line.

Singular shows this part when help is requested for the relative library, that is

```
help 123456.lib;
```

It should also be noted that strings are allocated to `version` and `info` by means of the sign of equality, `=`, so that the inverted commas `"`, which box them in, are just as necessary as the semicolon at the end of the line!

Section two serves the loading from other libraries, whose procedures are necessary for one's own procedures. As an example, the library `inout.lib`, whose procedure `pmat` in the `example` part of the procedure `permcol` is used.

In the third section the procedures follow, simply strung together. (It should be noted that the command `proc` is always shown at the beginning of a new line!) It is recommended that the Syntax in section 2.1.7 is extended by two sections for procedures.

A commentary block can be inserted between the procedure head and the text body, in inverted commas `"`, which contains certain key words followed by the relative information. Under `USAGE:` should be shown how the command is summonsed and of which type the arguments are. `RETURN:` should contain information on which type the return is and, if necessary, any further information. `NOTE:` is used to show important comments to the procedure, its use, etc. `EXAMPLE:` shows tips as to how an example can be shown under Singular. The commentary block displayed there contains information which is shown when help is requested to a procedure under Singular, z.B. through

```
help permcol;
```

The second additional section at the end of the procedure is initiated through the set word `example`, followed by a section in curly brackets which contains the Singular code. The aim is to show an example for the operation of the procedure which simplifies its use for the user. The user obtains the example by entering `example PROCEDURENAME;`.

2.1.9 Output in files / input in files

The command `write` offer the possibility to store the values of variables or any strings in a file. For this, the variable values are converted to strings. The following lines store variable values, resp. a string, in the file `test1.txt`:

```
int a=5;
int b=4;
write("test1.txt",a,b);
write("test1.txt","Das ist Singular.");
```

Several variables or strings can be shown, separated by commas. Their values are written in each case in a new line.

Data contained in a file can be read in by the command `read`. They are, however, interpreted as strings, e.g.

```
read("test1.txt");
//-> 5
     4
     Das ist Singular.
```

Should Singular code, which is read in from a file, be recognized as such, then the `read` command must be passed on to the command `execute`. Should the file `test2.txt` contain the following lines,

```
4*5-3;
6^3;
```

which is to be done via

```
write("test2.txt","4*5-3;"+"6^3;");
read("test2.txt");
//-> 4*5-3;6^3;
```

then the command

```
execute(read("test2.txt"));
```

leads to the following Singular output:

```
//-> 17
     216
```

A short form for `execute(read(...))` is <, e.g.

```
< "test2.txt";
```

Anyone wanting to document a Singular session for security in a file, e.g. `test3.txt`, can do this with the command `monitor`, e.g.

```
monitor("test3.txt","io");
```

The option "io" causes input as well as output to be stored. The omission of one of the letters leads to only the input or only the output being stored. The option monitor is very helpful when working on an operating system on which SINGULAR is instable or on which an editor is available, which is no easily manageable.

Please note that monitor opens a file, but does not terminate it. This can be done by the following input:

```
monitor("");
```

2.2 Ringindependent objects

As we already mentioned in section 2.1.2, once SINGULAR is started, it awaits an input after the prompt >. Every statement has to be terminated by ;

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

Integer variables are ringindependent objects that are defined by the word int. To keep in mind, the assignment is done by the symbol = .

```
int k = 2;
```

In addition there is a bigint expression available in order to define a larger integer variable.

```
// Note: 11*13*17*100*200*2000
// returns a machine integer:
11*13*17*100*200*2000;
//-> // ** int overflow(*), result may be wrong
//-> -1544247808
// using the type cast number for a greater allowed range
bigint(11)*13*17*100*200*2000;
//-> 97240000000
```

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[2,3];
//-> 6
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.

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

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

```
"example for strings:";
//-> example for strings:

string s="The element of m ";
s = s + "at position [2,3] is:"; //concatenation of strings by +
s , m[2,3] , ".";
//-> The element of m at position [3,2] is: 6 .

s="m[2,1]=0; m;";
execute(s);
//-> 1,0,3,
//-> 0,5,6,
//-> 7,8,9
```

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

The following operators are given for the data type `list`.

`+`	Combines the elements of two lists.
`delete`	Deletes an element from a list, `delete(L,3)` deletes the third element of the list `L` (the input is not changed).
`insert`	Inserts an element into a list. `insert(L,4)` inserts into the list `L`, the element `4` in first place, `insert(L,4,2)` after second position, i.e. in third place (the input is not changed).

Here is a simple example:

```
string t = "arbitrary element";
intvec v = 1,2,3;
list L1 = 23,t,3;
list L2 = v,45;
L1;
//-> [1]:
//->    23
//-> [2]:
//->    arbitrary element
//-> [3]:
//->    3
```

```
L2;
//-> [1]:
//->    1,2,3
//-> [2]:
//->    45

list L = L1 + L2;
L;
//-> [1]:
//->    23
//-> [2]:
//->    arbitrary element
//-> [3]:
//->    3
//-> [4]:
//->    1,2,3
//-> [5]:
//->    45

delete(L,3);
//-> [1]:
//->    23
//-> [2]:
//->    arbitrary element
//-> [3]:
//->    1,2,3
//-> [4]:
//->    45

insert(L,4);
//-> [1]:
//->    4
//-> [2]:
//->    23
//-> [3]:
//->    arbitrary element
//-> [4]:
//->    3
//-> [5]:
//->    1,2,3
//-> [6]:
//->    45

insert(L,4,2);
//-> [1]:
//->    23
//-> [2]:
//->    arbitrary element
//-> [3]:
//->    4
```

```
//-> [4]:
//->    3
//-> [5]:
//->    1,2,3
//-> [6]:
//->    45

L;
//-> [1]:
//->    23
//-> [2]:
//->    arbitrary element
//-> [3]:
//->    3
//-> [4]:
//->    1,2,3
//-> [5]:
//->    45
```

Note that lists are not always ringindependent. Lists containing an element from a ring belong to that ring.

2.3 Rings and standard bases

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

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

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

Other ring declarations:

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

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

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

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

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

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

Defining a ring makes this ring the current active basering, so each ring definition above switches to a new basering. The concept of rings in SINGULAR is discussed in detail in the chapter "Rings and orderings" of the SINGULAR manual [GPS1].

Furthermore one can define a ring `r6` which extends the ring `r` by adding new variables in front of the old variables.

```
LIB "ring.lib";
def r6 = extendring(3,"t(","dp");
```

The characteristic of `r6` is the characteristic of `r`, the new variables are `t(1),t(2),t(3)` and the ordering is the product ordering of the ordering of `r` and `dp`.
The basering now is `r5`. That means to work in the ring `r6` we have to switch to it. This can be done using the function `setring`:

```
setring r6;
r6;
//-> //   characteristic : 0
//-> //   number of vars : 6
//-> //        block   1 : ordering dp
//-> //                  : names    t(1) t(2) t(3)
//-> //        block   2 : ordering dp
//-> //                  : names    x y z
//-> //        block   3 : ordering C
```

Since we want to calcualate in the ring `r`, which we defined first, we have to switch back to it.

```
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,x,1,y,2,z,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 calculation, if desired. The command `option(prot);` enables protocoling whereas `option(noprot);` turns it off.

Singular's implementation of Buchberger's algorithm is available via the `std` command (`std` referring to standard basis). The computation of reduced Groebner and standard bases may be forced by setting `option(redSB)`. However, depending on the monomial ordering of the active basering, it may be advisable to use the `groebner` command instead. This command is provided by the Singular library `standard.lib` which is automatically loaded when starting a Singular session. Depending on some heuristics, `groebner` either refers to the `std` command (e.g. for rings with ordering `dp`), or to one of the other implemented Groebner bases algorithms. For information on the heuristics behind `groebner`, see the library file `standard.lib`.

```
ideal I1 = x3+y2,xyz-y2;
std(I1);
//-> _[1]=xyz-y2
//-> _[2]=x3+y2
//-> _[3]=x2y2+y3z
//-> _[4]=y3z2+xy3
```

```
groebner(I1);
//-> _[1]=xyz-y2
//-> _[2]=x3+y2
//-> _[3]=x2y2+y3z
//-> _[4]=y3z2+xy3
```

The same generators for an ideal give different standard bases with respect to different orderings:

```
ring A  = 0,(x,y),dp;   //global ordering: degrevlex
ideal I2 = x10+x9y2,y8-x2y7;
ideal sI2 = std(I2);
sI2;
//-> sI2[1]=x2y7-y8   sI2[2]=x9y2+x10   sI2[3]=x12y+xy11
//-> sI2[4]=x13-xy12  sI2[5]=y14+xy12   sI2[6]=xy13+y12

ring A1 = 0,(x,y),lp;   //global ordering: lex
ideal I3 = fetch(A,I2);
ideal sI3 = std(I3);
sI3;
//-> sI3[1]=y15-y12
//-> sI3[2]=xy12+y14
//-> sI3[3]=x2y7-y8
//-> sI3[4]=x10+x9y2
```

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 [GPS1] Critical points, for a detailed explanation.

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

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

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

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

```
J = groebner(J);
ideal kJ = kbase(J);
kJ;
//-> kJ[1]=y4
//-> kJ[2]=xy3
//-> kJ[3]=y3
```

```
//-> kJ[4]=xy2
//-> kJ[5]=y2
//-> kJ[6]=x2y
//-> kJ[7]=xy
//-> kJ[8]=y
//-> kJ[9]=x3
//-> kJ[10]=x2
//-> kJ[11]=x
//-> kJ[12]=1
```

Then

```
size(kJ);
//-> 12
```

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

```
size(kbase(groebner(jacob(f))));
//-> 12
```

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

```
vdim(groebner(jacob(f)));
//-> 12
```

3 Examples

3.1 Computation in fields

In Singular, field elements have the type `number` but notice that in order to compute in fields, i.e. to *define and use numbers one has to define a polynomial ring with at least one variable* and a specified monomial ordering.

(a) Computation in the field of *rational numbers*:

```
ring A = 0,x,dp;
number n = 12345/6789;
n^5;                    //common divisors are cancelled
//-> 1179910858126071875/59350279669807543
```

Note: Typing just `123456789^5;` will result in integer overflow since `123456789` is considered as an integer (machine integer of limited size) and not as an element in the field of rational numbers; however, also correct would be `number(123456789)^5;`.

```
int a = 5;
int b = 3;
a;
//-> 5
a+b;
//-> 8
a-b;
//-> 2
a/b;
//-> 1
a%b;
//-> 2
a*b;
//-> 15
a^b;
//-> 125
5/3;
//-> 5/3
number c = 5;
number d = 3;
c/d;
//-> 5/3
```

(b) Computation in *finite fields*:

```
ring A1 = 32003,x,dp;   //finite field Z/32003
number(123456789)^5;
//-> 8705
number a = 25000;
number b = 20000;
a+b;
//-> 12997
a/b;
//-> 8002

ring A2 = (2^3,a),x,dp; //finite (Galois) field GF(8)
                        //with 8 elements
number n = a+a2;        //a is a generator of the group
                        //GF(8)-{0}
n^5;
//-> a6
minpoly;                //minimal polynomial of GF(8)
//-> 1*a^3+1*a^1+1*a^0

ring A3 = (2,a),x,dp;   //infinite field Z/2(a) of
                        //characteristic 2
minpoly = a20+a3+1;     //define a minimal polynomial
                        //a^20+a^3+1
                        //now the ground field is
                        //GF(2^20)=Z/2[a]/<a^20+a^3+1>,
number n = a+a2;        //a finite field
                        //with 2^20 elements
n^5;                    //a is a generator of the group
                        //GF(2^20)-{0}
//-> (a10+a9+a6+a5)
```

Note: For computation in finite fields $\mathbb{Z}/p\mathbb{Z}$, $p \leq 32003$, respectively $GF(p^n)$, $p^n \leq 2^{15}$, one should use rings as `A1` respectively `A2` since for these fields Singular uses look–up tables, which is quite fast. For other finite fields a *minimal polynomial* as in `A3` must be specified. Singular does not, however, check the irreducibility of the chosen minimal polynomial. This can be done as in the following example.

```
ring tst = 2,a,dp;
factorize(a20+a2+1,1);
//-> _[1]=a3+a+1     //not irreducible! We have 2 factors
//-> _[2]=a7+a5+a4+a3+1
factorize(a20+a3+1,1);   //irreducible
//-> _[1]=a20+a3+1
```

To obtain the multiplicities of the factors, use `factorize(a20+a2+1);`

(c) Computation with *real* and *complex floating point numbers*, 30 digits precision:

```
ring R1 = (real,30),x,dp;
number n = 123456789.0;
```

```
n^5;          //compute with a precision of 30 digits
//-> 0.286797186029971810723376143809e+41
```

Note: n^5 is a number whose integral part has 41 digits (indicated by e+41). However, only 30 digits are computed.

```
ring R2 = (complex,30,I),x,dp; //I denotes
                               //imaginary unit
number n = 123456789.0+0.0001*I;
n^5;                           //complex number with
                               //30 digits precision
//-> (0.286797186029971810723374262133e+41
     +I*116152861399129622075046746710)
```

(d) Computation with rational numbers and *parameters*, that is, in $\mathbb{Q}(a,b,c)$, the quotient field of $\mathbb{Q}[a,b,c]$:

```
ring R3 = (0,a,b,c),x,dp;
number n = 12345a+12345/(78bc);
n^2;
//->(103021740900a2b2c2+2641583100abc+16933225)/(676b2c2)
n/9c;
//-> (320970abc+4115)/(234bc2)
```

3.2 Computation in polynomial rings

We shall now show how to define the polynomial ring in n variables $x_1,\ldots,x_n$ over the above mentioned fields K. We can do this for any n, but we have to specify an integer n first. The same remark applies if we work with transcendental extensions of degree m; we usually call the elements $t_1,\ldots,t_m$ of a transcendental basis (free) *parameters*. If g is any non–zero polynomial in the parameters $t_1,\ldots,t_m$, then g and $1/g$ are numbers in the corresponding ring. For further examples see [GPS1].

(a) Computation with *polynomials*:

Let us create *polynomial rings* over different fields. By typing the name of the *ring* we obtain all relevant information about the ring.

```
ring A = 0,(x,y,z),dp;
poly f = x3+y2;           //same as x^3+y^2
f*f-f;
//-> x6+2x3y2+y4-x3-y2
```

Singular understands short (e.g., 2x2+y3) and long (e.g., 2*x^2+y^3) input. Computations in polynomial rings over other fields follow the same pattern. Try ring R=32003,x(1..3),dp; (finite ground field), and type

`R;` to obtain information about the ring. The command `setring` allows switching from one ring to another, for example, `setring A;` makes `A` the basering.

```
poly g = 2xy-z2;
f+g;
//-> x3+2xy+y2-z2
f*g;
//-> 2x4y-x3z2+2xy3-y2z2
```

(b) Declaration and operation on *ideals*:

```
ideal I = 0,x,0,1;
I;
//-> I[1]=0
//-> I[2]=x
//-> I[3]=0
//-> I[4]=1
I + 0;      // addition with 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

ideal m1 = maxideal(1);
m1^2;       // exponentiation
//-> _[1]=x2
//-> _[2]=xy
//-> _[3]=xz
//-> _[4]=y2
//-> _[5]=yz
//-> _[6]=z2
ideal m2 = maxideal(2);
m2;
//-> _[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
```

3.3 Methods for creating ring maps

SINGULAR has three possibilities, `fetch`, `imap` and `map`, to define ring maps by giving the name of the preimage ring and a list of polynomials $f_1, \dots, f_n$ (as many as there are variables in the preimage ring) in the current basering. The commands `fetch`, respectively `imap`, map an object directly from the preimage ring to the basering whereas `fetch` maps the first variable to the first, the second to the second and so on (hence, is convenient for renaming the variables), while `imap` maps a variable to the variable with the same name (or to 0 if it does not exist), hence is convenient for inclusion of sub–rings or for changing the monomial ordering.

Note: All maps go from a predefined ring to the basering.

$$\text{map: preimage ring} \longrightarrow \text{basering}$$

(a) General definition of a *map*:

```
ring A = 0,(a,b,c),dp;
poly f = a+b+ab+c3;
f;
//-> c3+ab+a+b

ring B = 0,(x,y,z),dp;
map F  = A, x+y,x-y,z; //map F from ring A (to
                       //basering B) sending
                       //a -> x+y, b -> x-y, c -> z
poly g = F(f);         //apply F
g;
//-> z3+x2-y2+2x
```

(b) Special maps (`imap,fetch`):

```
ring A1 = 0,(x,y,c,b,a,z),dp;
imap(A,f);        //imap preserves names of variables
//-> c3+ba+b+a
fetch(A,f);       //fetch preserves order of variables
//-> c3+xy+x+y
```

3.4 Properties of ring maps

(1) Checking injectivity:

```
ring S = 0,(a,b,c),lp;
ring R = 0,(x,y,z),dp;
ideal i = x, y, x2-y3;
map phi = S,i;            //a map from S to R, sending
                          //a -> x, b -> y, c -> x2-y3
LIB "algebra.lib";        //load algebra.lib
```

By default, SINGULAR displays the names and paths of those libraries which are used by `algebra.lib` and which are also loaded. We suppress this message.

We test injectivity using the procedure `is_injective`, then we compute the kernel by using the procedure `alg_kernel` (which displays the kernel, an object of the preimage ring, as a string).

```
is_injective(phi,S);
//-> 0                        // phi is not injective
ideal j = x, x+y, z-x2+y3;
map psi = S,j;                // another map from S to R
is_injective(psi,S);
//-> 1                        // psi is injective
alg_kernel(phi,S);
//-> b^3-a^2+c                // <b^3-a^2+c> = Ker(phi)
alg_kernel(psi,S);
//-> 0
```

(2) Computing the *preimage*:

Using the `preimage` command, we must first go back to S, since the preimage is an ideal in the preimage ring.

```
ideal Z;                 //the zero ideal in R
setring S;
preimage(R,phi,Z);       //computes kernel of phi in S
//-> _[1]=a2-b3-c        //kernel of phi = preimage of Z
```

(3) Checking *surjectivity* and *bijectivity*:

```
setring R;
is_surjective(psi,S);
//-> 1
is_bijective(psi,S);        //faster than is_injective,
                            //is_surjective
//-> 1
```

(4) Computing the *inverse* in quotient rings:

If $I \subset K[x] = K[x_1, \dots, x_n]$ is a maximal ideal, then the quotient ring $K[x]/I$ is a field. To be able to compute effectively in the field $K[x]/I$ we need, in addition to the ring operations, the inverse of a non–zero element. The following example shows that we can effectively compute in all fields of finite type over a prime field.

If the polynomial f is invertible, then the command `lift(f,1)[1,1]` gives the inverse (`lift` checks whether $1 \in \langle f \rangle$ and then expresses 1 as a multiple of f):

```
ring R=(0,x),(y,z),dp;
ideal I=-z5+y2+(x2),-y2+z+(-x);
I=std(I);
qring Q=I;
```

We shall now compute the inverse of z in $Q = R/I$.

```
poly p=lift(z,1)[1,1];
p;
//-> 1/(x2-x)*z4-1/(x2-x)
```

We make a test for p being the inverse of z.

```
reduce(p*z,std(0));
//-> 1
```

The ideal I is a maximal ideal if and only if R/I is a field. We shall now prove that, in our example, I is a maximal ideal.

```
ring R1=(0,x),(z,y),lp;
ideal I=imap(R,I);
I=std(I);
I;
//-> I[1]=y10+(5x)*y8+(10x2)*y6+(10x3)*y4+(5x4-1)*y2+(x5-x2)
//-> I[2]=z-y2+(-x)
```

Since $\mathbb{Q}(x)[z,y]/\langle z - y^2 - x\rangle \cong \mathbb{Q}(x)[y]$, we see that

$$R/I \cong \mathbb{Q}(x)[y]/\langle y^{10} + 5xy^8 + 10x^2y^6 + 10x^3y^4 + (5x^4 - 1)y^2 + x^5 - x^2\rangle .$$

```
factorize(I[1]);
//-> [1]:
//->     _[1]=1
//->     _[2]=y10+(5x)*y8+(10x2)*y6+(10x3)*y4+(5x4-1)*y2
//->              +(x5-x2)
//-> [2]:
//->     1,1
```

The polynomial $I[1]$ is irreducible and R/I is a field.

3.5 Monomial orderings

Global orderings are denoted with a `p` at the end, referring to "polynomial ring" while local orderings end with an `s`, referring to "series ring". Note that Singular stores and outputs a polynomial in an ordered way, in decreasing order.

(a) Global orderings:

```
ring A1 = 0,(x,y,z),lp;        //lexicographical
poly f = x3yz + y5 + z4 + x3 + xy2; f;
//-> x3yz+x3+xy2+y5+z4

ring A2 = 0,(x,y,z),dp;        //degree reverse
                               //lexicographical
poly f = imap(A1,f); f;
//-> y5+x3yz+z4+x3+xy2

ring A3 = 0,(x,y,z),Dp;        //degree lexicographical
poly f = imap(A1,f); f;
//-> x3yz+y5+z4+x3+xy2

ring A4 = 0,(x,y,z),Wp(5,3,2); //weighted degree
                               //lexicographical
poly f = imap(A1,f); f;
//-> x3yz+x3+y5+xy2+z4
```

(b) Local orderings:

```
ring A5 = 0,(x,y,z),ls;        //negative lexicographical
poly f = imap(A1,f); f;
//-> z4+y5+xy2+x3+x3yz

ring A6 = 0,(x,y,z),ds;        //negative degree reverse
                               //lexicographical
poly f = imap(A1,f); f;
//-> x3+xy2+z4+y5+x3yz

ring A7 = 0,(x,y,z),Ws(5,3,2); //negative weighted degree
                               //lexicographical
poly f = imap(A1,f); f;
//-> z4+xy2+x3+y5+x3yz
```

(c) Product and matrix orderings:

```
ring A8 = 0,(x,y,z),(dp(1),ds(2)); //mixed product ordering
poly f = imap(A1,f); f;
//-> x3+x3yz+xy2+z4+y5
```

```
intmat A[3][3] = -1, -1, -1, 0, 0, 1, 0, 1, 0;
print(A);
//->     -1    -1    -1
//->      0     0     1
//->      0     1     0
```

Now define your own matrix ordering using A:

```
ring A9 = 0,(x,y,z),M(A); //a local ordering
poly f = imap(A1,f); f;
//-> xy2+x3+z4+x3yz+y5
```

3.6 Leading data

```
ring A = 0,(x,y,z),lp;
poly f = y4z3+2x2y2z2+3x5+4z4+5y2;
f;                          //display f in a lex-ordered way
//-> 3x5+2x2y2z2+y4z3+5y2+4z4
leadmonom(f);               //leading monomial
//-> x5
leadexp(f);                 //leading exponent
//-> 5,0,0
lead(f);                    //leading term
//-> 3x5
leadcoef(f);                //leading coefficient
//-> 3
f - lead(f);                //tail
//-> 2x2y2z2+y4z3+5y2+4z4
```

3.7 Normal form

Note that $\mathrm{NF}(f \mid G)$ may depend on the sorting of the elements of G. The function `reduce` computes a normal form.

```
ring A  = 0,(x,y,z),dp; //a global ordering
poly f  = x2yz+xy2z+y2z+z3+xy;
poly f1 = xy+y2-1;
poly f2 = xy;
ideal G = f1,f2;
ideal S = std(G);       //a standard basis of <G>
S;
//-> S[1]=x
//-> S[2]=y2-1

reduce(f,G);
//** G is no standardbasis
//-> y2z+z3             //NF w.r.t. a non-standard
                        //basis
```

```
G=f2,f1;
reduce(f,G);
//** G is no standardbasis
//-> y2z+z3-y2+1        //NF for a different numbering
                        //in G
reduce(f,S,1);          //NFBuchberger
//-> z3+xy+z

reduce(f,S);            //redNFBuchberger
//-> z3+z
```

3.8 Ideal membership

(1) Check inclusion of a polynomial in an ideal:

```
ring A  = 0,(x,y),dp;
ideal I = x10+x9y2,y8-x2y7;
ideal sI = std(I);
poly f  = x2y7+y14;
reduce(f,sI,1);     //NFBuchberger, 3rd parameter avoids
                    //tail reduction
//-> -xy12+x2y7     //f is not in I
NF(f,sI,1);
//-> -xy12+x2y7     //f is not in I
f  = xy13+y12;
reduce(f,sI,1);
//-> 0              //f is in I
```

(2) Check inclusion and equality of ideals:

```
ideal K = f,x2y7+y14;
reduce(K,sI,1);               //normal form for
                              //each generator of K
//-> _[1]=0  _[2]=-xy12+x2y7  //K is not in I

K=f,y14+xy12;
size(reduce(K,sI,1));         //result is 0 iff K is in I
//-> 0
```

3.9 Linear combination of ideal members

Now assume that $f \in I = \langle f_1, \dots, f_k \rangle$. Then there exist $a_1, \dots, a_k \in K[x]$ such that

$$f = a_1 f_1 + \cdots + a_k f_k.$$

The computation of the a_i is illustrated in the following example.
We exemplify the Singular commands `lift` and `division`:

```
ring A  = 0,(x,y),dp;
ideal I = x10+x9y2,y8-x2y7;
poly f  = xy13+y12;
matrix M=lift(I,f);          //f=M[1,1]*I[1]+...+M[r,1]*I[r]
M;
//-> M[1,1]=y7
//-> M[2,1]=x7y2+x8+x5y3+x6y+x3y4+x4y2+xy5+x2y3+y4
```

Hence, f can be expressed as a linear combination of $I[1]$ and $I[2]$ using M:

```
f-M[1,1]*I[1]-M[2,1]*I[2];   //test
//-> 0
```

3.10 Elimination of variables

```
ring A =0,(t,x,y,z),dp;
ideal I=t2+x2+y2+z2,t2+2x2-xy-z2,t+y3-z3;

eliminate(I,t);
//-> _[1]=x2-xy-y2-2z2      _[2]=y6-2y3z3+z6+2x2-xy-z2

LIB "elim.lib";   //loads library for elim1
elim1(I,t);
//-> _[1]=x2-xy-y2-2z2      _[2]=y6-2y3z3+z6+2x2-xy-z2
```

Alternatively choose a product ordering:

```
ring A1=0,(t,x,y,z),(dp(1),dp(3));
ideal I=imap(A,I);
ideal J=std(I);
J;
//-> J[1]=x2-xy-y2-2z2    J[2]=y6-2y3z3+z6+2x2-xy-z2
//-> J[3]=t+y3-z3
```

3.11 Computing with radicals

(1) Compute the *radical* of an ideal:

```
ring R = 0,(x,y,z),dp;
poly p = z4+2z2+1;
LIB "primdec.lib";        //loads library for radical

radical(p);               //squarefree part of p
//-> _[1]=z2+1

ideal I = xyz, x2, y4+y5; //a more complicated ideal
radical(I);
//-> _[1]=x
//-> _[2]=y2+y            //we see that I is not reduced
```

(2) Compute the *index of nilpotency*:

Since $y^2 + y$ is contained in the radical of I, some power of $y^2 + y$ must be contained in I. We compute the minimal power k so that $(y^2 + y)^k$ is contained in I by using the normal form. This is the same as saying that $y^2 + y$ is nilpotent in the quotient ring R/I and then k is the index of nilpotency of $y^2 + y$ in R/I.

```
ideal gI = groebner(I);
int k;
while (reduce((y2+y)^k,gI) != 0 ) {k++;}
k;
//-> 4       //minimal power (index of nilpotency) is 4
```

3.12 Intersection of ideals

We use elimination to compute the intersection of two ideals:
Given $f_1, \dots, f_k$, $h_1, \dots, h_r \in K[x]$ and $>$ a monomial ordering.
Let $I_1 = \langle f_1, \dots, f_k \rangle K[x]$ and $I_2 = \langle h_1, \dots, h_r \rangle K[x]$. We wish to find generators for $I_1 \cap I_2$.

Consider the ideal $J := \langle tf_1, \dots, tf_k, (1-t)h_1, \dots, (1-t)h_r \rangle (K[x])[t]$.
With the above notations we get $I_1 \cap I_2 = J \cap K[x]$, i.e. we obtain the intersection by eliminating t from J.

```
ring A=0,(x,y,z),dp;
ideal I1=x,y;
ideal I2=y2,z;
intersect(I1,I2);         //the built-in SINGULAR command
//-> _[1]=y2    _[2]=yz    _[3]=xz

ring B=0,(t,x,y,z),dp;    //the way described above
ideal I1=imap(A,I1);
ideal I2=imap(A,I2);
ideal J=t*I1+(1-t)*I2;
eliminate(J,t);
//-> _[1]=yz  _[2]=xz  _[3]=y2

ideal I3 = x;
ideal I4 = y;
intersect(I3,I4);
//-> _[1]=xy
```

3.13 Quotient of ideals

Let I_1 and $I_2 \subset K[x]$. We want to compute

$$I_1 : I_2 = \{g \in K[x] \mid gI_2 \subset I_1\}.$$

Therefore the procedure `quotient` is implemented in SINGULAR.

```
ring A=0,(x,y,z),dp;
ideal I1=x,y;
ideal I2=y2,z;
quotient(I1,I2);     //the built-in SINGULAR command
//-> _[1]=y    _[2]=x

ideal I3 = xz,yz;
ideal I4 = x,y;
quotient(I3,I4);
//-> _[1]=z
```

3.14 Matrix operations

A matrix in SINGULAR is a matrix with polynomial entries, hence they can be defined only when a basering is active. This applies also to matrices with numbers as entries. A matrix is filled with entries from left to right, row by row, spaces are allowed.

```
ring A = 0,(x,y,z),dp;
matrix M[2][3] = 1, x+y, z2,        //2x3 matrix
                 x,   0, xyz;
matrix N[3][3] = 1,2,3,4,5,6,7,8,9; //3x3 matrix

M;                                  //lists all entries of M
//-> M[1,1]=1
//-> M[1,2]=x+y
//-> M[1,3]=z2
//-> M[2,1]=x
//-> M[2,2]=0
//-> M[2,3]=xyz

print(N);                 //displays N as usual
//-> 1,2,3,               //if the entries are small
//-> 4,5,6,
//-> 7,8,9

print(M+M);               //addition of matrices
//-> 2, 2x+2y,2z2,
//-> 2x,0,    2xyz

print(x*N);
//-> x, 2x,3x,            //scalar multiplication
//-> 4x,5x,6x,
//-> 7x,8x,9x

print(M*N);               //multiplication of matrices
//-> 7z2+4x+4y+1,8z2+5x+5y+2,9z2+6x+6y+3,
//-> 7xyz+x,     8xyz+2x,    9xyz+3x
```

```
ideal I = minor(M,2);     //ideal of all 2x2 minors (sub-
                          //determinants) of M
I;
//-> I[1]=x2yz+xy2z
//-> I[2]=xyz-xz2
//-> I[3]=-x2-xy
minor(N,2);               //2x2 minors of N
//-> _[1]=-3
//-> _[2]=-6
//-> _[3]=-6
//-> _[4]=-12
//-> _[5]=3
//-> _[6]=6
//-> _[7]=-3
//-> _[8]=-6
//-> _[9]=3

M[2,3];                   //access to single entry
//-> xyz
M[2,3]=37;                //change single entry
print(M);
//-> 1,x+y,z2,
//-> x,0,  37
```

Further matrix operations are contained in the library `matrix.lib`. There is a procedure `pmat` in `inout.lib` which formats matrices similarly to `print`, but allows additional parameters, for example to show only the first terms of each entry for big matrices.

```
LIB "matrix.lib";
LIB "inout.lib";

print(power(N,3));        //exponentiation of matrices
//-> 468, 576, 684,
//-> 1062,1305,1548
//-> 1656,2034,2412

pmat(power((x+y+z)*N,3),15);  //show first 15 terms of entries
                              //where a truncation of a column
                              //is indicated by two dots
//-> 468x3+1404x2y.., 576x3+1728x2y.., 684x3+2052x2y..,
//-> 1062x3+3186x2.., 1305x3+3915x2.., 1548x3+4644x2..,
//-> 1656x3+4968x2.., 2034x3+6102x2.., 2412x3+7236x2..

matrix K = concat(M,N);   //concatenation
print(K);
//-> 1,x+y,z2,1,2,3,
//-> x,0,  37,4,5,6,
//-> 0,0,  0, 7,8,9
```

```
ideal(M);                  //converts matrix to ideal
//-> _[1]=1                //same as ‘flatten’ from matrix.lib
//-> _[2]=x+y
//-> _[3]=z2
//-> _[4]=x
//-> _[5]=0
//-> _[6]=37

print(unitmat(5));         //5x5 unit matrix
//-> 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,
```

Besides matrices, there are integer matrices which do not need a ring. These are mainly used for bookkeeping or storing integer results. The operations are the same as for matrices.

```
intmat IM[2][3]=1,2,3,4,5,6;
IM;
//-> 1,2,3,
//-> 4,5,6
print(IM);
//->    1     2     3
//->    4     5     6
```

3.15 Langrange multipliers

3.15.1 Theoretical introduction

In many optimization problems we do not only ask for the extremum of a function, but for the extremum under additional conditions.
Lets consider the following problem: Given a function $f : U \longrightarrow \mathbb{R}$ and functions $\varphi_1, \dots, \varphi_k : U \longrightarrow \mathbb{R}$ defined on a set $U \subset \mathbb{R}^n$. Let M be the zero-set of $\varphi = (\varphi_1, \dots, \varphi_k) : U \longrightarrow \mathbb{R}^k$:

$$M = \{x \in U \mid \varphi(x) = 0\}.$$

We are interested in local extrema x_0 of f on M, i.e.

$$\begin{aligned} f(x) &\leq f(x_0) \quad \forall\, x \in M, \text{ or} \\ f(x) &\geq f(x_0) \quad \forall\, x \in M. \end{aligned}$$

Such points are called *Maxima resp. Minima of f on M* or *Maxima resp. Minima of f under the additional condition $\varphi = 0$*. The following theorem states a necessary condition for extrema of f on M, if M is a manifold.

Theorem 3.15.1 (Lagrange)**.** *Let $f : U \longrightarrow \mathbb{R}$ be differentiable and $\varphi = (\varphi_1, \dots, \varphi_k) : U \longrightarrow \mathbb{R}$ be continuously differentiable on an open set $U \subset \mathbb{R}^n$. Furthermore let the Jacobian matrix of φ have rank k at any point x of the zero-set M of φ, i.e.*

$$rk(J(\varphi)(x)) = k \quad \forall\, x \in M = \{x \in U \mid \varphi(x) = 0\}.$$

Then it holds:
If $x_0 \in M$ is an extremum of f, then $\nabla f(x_0)$ is a linear combination of $\nabla\varphi_1(x_0), \dots, \nabla\varphi_k(x_0)$, i.e. there exist $\lambda_1, \dots, \lambda_k \in \mathbb{R}$ satisfying

$$\nabla f(x_0) = \sum_{i=1}^{k} \lambda_i \nabla\varphi_i(x_0).$$

The numbers $\lambda_1, \dots, \lambda_k$ are called Lagrange Multipliers.

Proof. For the proof we refer to [KOE]. □

3.15.2 Application to Singular

Let $\mathbb{Q}[x] = \mathbb{Q}[x_1, \dots, x_n]$ be the polynomial ring in n variables over the field $\mathbb{Q}$ and consider an ideal $G = \langle g_1, \dots, g_k \rangle \subset \mathbb{Q}[x_1, \dots, x_n]$ and a polynomial $f \in \mathbb{Q}[x_1, \dots, x_n]$.
We want to compute local extrema of the map $f|_M : M \longrightarrow \mathbb{Q}$ where M is a manifold given by

$$M := V(G) := V(g_1, \dots, g_k) = \{x \in \mathbb{Q}^n \mid g_1(x) = \ldots = g_k(x) = 0\} \subset \mathbb{Q}^n,$$

i.e. M denotes the set of solutions of the polynomial system

$$\begin{array}{rcl} g_1(x_1, \dots, x_n) & = & 0 \\ \vdots & \vdots & \vdots \\ g_k(x_1, \dots, x_n) & = & 0 \end{array}$$

and is called the (<u>affine algebraic</u>) <u>variety</u> of $g_1, \dots, g_k$, respectively of G.

Definition 3.15.2. An affine algebraic variety $M = V(g_1, \dots, g_k)$ is called an *algebraic manifold* if the Jacobian matrix

$$J(g_1, \dots, g_k)(p) = \begin{pmatrix} \frac{\partial g_1}{\partial x_1}(p) & \dots & \frac{\partial g_1}{\partial x_n}(p) \\ \vdots & & \vdots \\ \frac{\partial g_k}{\partial x_1}(p) & \dots & \frac{\partial g_k}{\partial x_n}(p) \end{pmatrix}$$

has maximal rank at all $p \in M$.

In terms of our assumptions and Definition 3.15.2, Theorem 3.15.1 concerning the Lagrange Multipliers can be reformulated as follows:

Theorem 3.15.3. *Let $f, g_1, \dots, g_k \in \mathbb{Q}[x_1, \dots, x_n]$, $M = V(g_1, \dots, g_k)$ be an algebraic manifold and consider the* Lagrange polynomial

$$F(x, \lambda) := f(x) + \sum_{i=1}^{k} \lambda_i \cdot g_i(x)$$

with $x = (x_1, \dots, x_n) \in \mathbb{Q}^n$ and $\lambda = (\lambda_1, \dots, \lambda_k) \in \mathbb{Q}^k$.

Then, if $p \in M$ is a local extremum of $f|_M$ there exists $\overline{\lambda} = (\overline{\lambda_1}, \dots, \overline{\lambda_k}) \in \mathbb{Q}^k$ such that the system of polynomial equations

$$\nabla F(x,\lambda) = 0 \iff \begin{array}{rcl} \nabla_x F(x,\lambda) & = & 0 \\ \nabla_\lambda F(x,\lambda) & = & 0 \end{array} \tag{3.1}$$

$$\iff \left\{ \begin{array}{rcl} \frac{\partial f}{\partial x_1}(x) + \sum_{i=1}^k \lambda_i \frac{\partial g_i}{\partial x_1}(x) & = & 0 \\ \vdots & \vdots & \vdots \\ \frac{\partial f}{\partial x_n}(x) + \sum_{i=1}^k \lambda_i \frac{\partial g_i}{\partial x_n}(x) & = & 0 \\ g_1(x) & = & 0 \\ \vdots & \vdots & \vdots \\ g_k(x) & = & 0 \end{array} \right\} \tag{3.2}$$

has the solution $(p, \overline{\lambda}) \in M \times \mathbb{Q}^k$.

In order to compute the local extrema of $f|_M : M \longrightarrow \mathbb{Q}$ we can therefore proceed in the following way:

- Eliminate $\lambda_1, \dots, \lambda_k$ from the system of polynomial equations (3.2).
- Get a system of equations $h_1(x) = \dots = h_r(x) = 0$ satisfying

$$\begin{array}{rcl} h_1(p) = \dots = h_r(p) = 0 & \iff & \exists\, \overline{\lambda} \text{ such that (3.2) holds} \\ & \Longleftarrow & f|_M \text{ has a local extremum at } p \in M \end{array}$$

SINGULAR Example 3.15.4. Compute the critical points of $f(x,y) = x^2 + y^2 \in \mathbb{Q}[x,y]$ under the additional condition $g(x,y) = 0$ where $g(x,y) = y - 2x + 1 \in \mathbb{Q}[x,y]$:

```
ring R = 0,(x,y),dp;
poly f = x2+y2;
ideal G = y-2x+1;
matrix jG = jacob(G);
jG;
//-> jG[1,1]=-2
//-> jG[1,2]=1
ideal M = minor(jG,1);
M;
//-> M[1]=1
//-> M[2]=-2
ideal I = M+G;
I;
//-> I[1]=1

LIB "ring.lib";
def r = extendring(1,"l","dp");
setring r;
poly f = imap(R,f);
ideal G = imap(R,G);
poly F = f+l*G[1];
```

```
F;
//-> -2lx+ly+l+x2+y2
ideal J = jacob(F);
J;
//-> J[1]=-2x+y+1
//-> J[2]=-2l+2x
//-> J[3]=l+2y
ideal crit = eliminate(J,l);
crit;
//-> crit[1]=5y+1
//-> crit[2]=2x-y-1

LIB "solve.lib";
setring R;
ideal crit = imap(r,crit);
solve(crit);
//-> [1]:
//->    [1]:
//->       0.4
//->    [2]:
//->       -0.2
```

SINGULAR Example 3.15.5. Check whether $V(g)$ with $g(x,y) = y^2 - x^3 - x^2 \in \mathbb{Q}[x,y]$ is an algebraic manifold:

```
ring R = 0,(x,y),dp;
ideal G = y2-x3-x2;
matrix jG = jacob(G);
jG;
//-> jG[1,1]=-3x2-2x
//-> jG[1,2]=2y
ideal I = jG + G;
I;
//-> I[1]=2y
//-> I[2]=-3x2-2x
//-> I[3]=-x3-x2+y2
std(I);
//-> _[1]=y
//-> _[2]=x

LIB "solve.lib";
solve(I);
//-> [1]:
//->    [1]:
//->       0
//->    [2]:
//->       0
```

Hence, $V(g)$ is not an algebraic manifold since $(0,0) \in V(g)$ is a singular point of g.

SINGULAR Example 3.15.6. Compute the critical points of $f(x,y) = x + y \in \mathbb{Q}[x,y]$ under the additional condition $g(x,y) = 0$ where $g(x,y) = \frac{1}{4}x^2 + \frac{1}{16}y^2 - 1 \in \mathbb{Q}[x,y]$:

```
ring R = 0,(x,y),dp;
poly f = x+y;
ideal G = 1/4*x2+1/16*y2-1;
matrix jG = jacob(G);
jG;
//-> jG[1,1]=1/2x
//-> jG[1,2]=1/8y
ideal I = jG+G;              // jG = minor(jG,1)
I;
//-> I[1]=1/2x
//-> I[2]=1/8y
//-> I[3]=1/4x2+1/16y2-1
std(I);
//-> _[1]=1
LIB "ring.lib";
def r = extendring(1,"l","dp");
setring r;
poly f = imap(R,f);
ideal G = imap(R,G);
poly F = f+l*G[1];
F;
//-> 1/4lx2+1/16ly2-l+x+y
ideal J = jacob(F);
J;
//-> J[1]=1/4x2+1/16y2-1
//-> J[2]=1/2lx+1
//-> J[3]=1/8ly+1
ideal crit = eliminate(J,l);
crit;
//-> crit[1]=4x-y
//-> crit[2]=5y2-64
LIB "solve.lib";
setring R;
ideal crit = imap(r,crit);
solve(crit);
//-> [1]:
//->    [1]:
//->       -0.89442719
//->    [2]:
//->       -3.57770876
//-> [2]:
//->    [1]:
//->       0.89442719
//->    [2]:
//->       3.57770876
```

3.15.3 Singular-Exercise

Consider an ideal $G = \langle g_1, \dots, g_k \rangle \subset \mathbb{Q}[x] = \mathbb{Q}[x_1, \dots, x_n]$ and a polynomial $f \in \mathbb{Q}[x] = \mathbb{Q}[x_1, \dots, x_n]$.

(1) Write a procedure which returns the Lagrange polynomial $F(x, \lambda) = f(x) + \sum_{i=1}^{k} \lambda_i \cdot g_i(x)$.

(2) How could you test with SINGULAR if the Jacobian matrix of $(g_1, \dots, g_k)$ has maximal rank on the zero-set of $(g_1, \dots, g_k)$?

(3) Write a procedure which returns the ideal of all critical points for the given data, if feasible.

3.15.4 Solution

(1) Here is just an example procedure to solve the given problem in SINGULAR:

```
LIB "ring.lib";     // necessary for procedure "extendring"
proc lagPoly(poly f, ideal G)
"USAGE: lagPoly(f,G);
RETURN: extended ring R with the variables
        (x(1),...,x(n),l(1),...,l(k)) comes with the Lagrange
        polynomial F(x,l)=f(x)+l(1)*g1(x)+...+l(k)*gk(x)
        where G=<g1,...,gk>
NOTE:   F is the Lagrange polynomial, whose partial deriviates
        are the Lagrange conditions;
        f is the function to maximize, G is the ideal defining
        the variety of points for which f is defined
"
{
  def S = basering;
  int nvar = size(G);
  def R = extendring(nvar, "l(", "dp");
  setring R;
  poly f = imap(S, f);
  poly F = f;
  ideal G = imap(S, G);
  for (int i = 1; i <= nvar; i++)
  {
    F = F + var(i) * G[i];
  }
  export(f);
  export(F);
  export(G);
  return(R);
}
```

(2) To observe if the Jacobian matrix $J(g_1, \dots, g_k) \in \mathrm{Mat}(k \times n, \mathbb{Q}[x])$ has full rank one can check whether there exists a $\big(n - dim(G)\big)$-*minor*[1] that does not vanish.

[1]By a *k-minor* we denote a determinant of a $k \times k$-submatrix of $A \in \mathrm{Mat}(n \times m, K)$.

Now consider $(g_1, \ldots, g_k) \in \mathbb{Q}[x]^k$, the matrix $J(g_1, \ldots, g_k) \in \mathrm{Mat}(k \times n, \mathbb{Q}[x])$ and let

$$M = \{m_i \mid m_i \text{ is a } \big(n - dim(G)\big)\text{-minor of } J(g_1, \ldots, g_k)\} \subset \mathbb{Q}[x].$$

Our aim is to obtain if

$$\forall\, p \in V(g_1, \ldots, g_k) : \;\; \exists\, i : \; m_i(p) \neq 0.$$

Therefore we define the ideal $I = \langle M, g_1, \ldots, g_k \rangle \subset \mathbb{Q}[x]$ and check if

$$V(I) = \emptyset \iff \begin{array}{l} 1 \in G = \{h_1, \ldots, h_l\} \subset I \\ \text{where } G \text{ is a Gröbner basis of } I. \end{array}$$

A possible realization for this in Singular is the following procedure:

```
proc lagTest(ideal G)
"USAGE: lagTest(G);
RETURN:  1, if the Jacobian matrix of G has maximal rank
            on V(G);
        -1, if the Jacobian matrix of G does not have
            maximal rank on V(G)
NOTE:   V(G) describes the variety/zero-set of the ideal
        G=<g1,...,gk>
"
{
  matrix jG = jacob(G);
  ideal M = minor(jG,nvars(basering)-dim(std(G)));
  ideal I = M + G;
  if(reduce(poly(1),std(I))==0)
  {
    return(1);
  }
  else
  {
    return(-1);
  }
}
```

(3) Since it is our aim to find all critical points of $f \in \mathbb{Q}[x] = \mathbb{Q}[x_1, \ldots, x_n]$ under the additional condition $g_1 = \ldots = g_k = 0$ we proceed as described in section 3.15.2. A possible realization in Singular serves the following procedure:

```
proc critIdeal(poly f, ideal G)
"USAGE: critIdeal(f,G);
RETURN: -1, if the Jacobian matrix of G does not have maximal
        rank on V(G) and else the ideal of all critical
        points p of f under the additional condition
        g1(p)=...=gk(p)=0
"
```

```
    {
      if(lagTest(G)==-1)
      {
        print("Attention: V(G) is no algebraic manifold!!");
        return(-1);
      }
      def S = basering;
      def R = lagPoly(f,G);
      setring R;
      poly L = 1;
      for (int i = 1; i <= nvars(R) - nvars(S); i++)
      {
        L = L * var(i);
      }
      ideal crit = eliminate(jacob(F), L);
      setring S;
      ideal crit = imap(R, crit);
      return(crit);
    }
```

SINGULAR Example 3.15.7. Compute the critical points of $f(x,y,z) = 2x+4y-5z \in \mathbb{Q}[x,y,z]$ under the additional condition $g(x,y,z) = (g_1(x,y,z), g_2(x,y,z)) = (0,0)$ where $(g_1(x,y,z), g_2(x,y,z)) = (x^2+y^2+z^2-16, -x-2y+z) \in \mathbb{Q}[x,y,z]^2$:

```
  ring R = 0,(x,y,z),dp;
  poly f = 2x+4y-5z;
  poly g1 = x2+y2+z2-16;
  poly g2 = -x-2y+z;
  ideal G = g1,g2;
  ideal I = critIdeal(f,G);
  I;
  //-> I[1]=5y-2z
  //-> I[2]=x+2y-z
  //-> I[3]=3z2-40
  solve(I);
  //-> [1]:
  //->    [1]:
  //->       -0.73029674
  //->    [2]:
  //->       -1.46059349
  //->    [3]:
  //->       -3.65148372
  //-> [2]:
  //->    [1]:
  //->       0.73029674
  //->    [2]:
  //->       1.46059349
  //->    [3]:
  //->       3.65148372
```

References

[GP] G.-M. Greuel, G. Pfister, *A* SINGULAR *Introduction to Commutative Algebra*, 2nd edition, Springer-Verlag, Berlin, 2007.

[GPS1] G.-M. Greuel, G. Pfister and H. Schönemann, SINGULAR *online manual.*

[GPS2] G.-M. Greuel, G. Pfister and H. Schönemann, SINGULAR 3-0-4 (2007), `http://www.singular.uni-kl.de`.

[DL] W. Decker, C. Lossen, *Computing in Algebraic Geometry*, Springer-Verlag, Berlin, 2006.

[KOE] K. Königsberger, *Analysis 2*, 2. erweiterte Auflage, Springer-Verlag, Berlin, 1997.

[MAR] T. Markwig, *A Short Introduction to Singular*, Lecture Notes, 2003.

FACHBEREICH MATHEMATIK, UNIVERSITÄT KAISERSLAUTERN, ERWIN-SCHRÖDINGER-STRASSE, D – 67663 KAISERSLAUTERN
E–mail address: steidel@mathematik.uni-kl.de
singular@mathematik.uni-kl.de to reach the SINGULAR team