# singular

## A GAP interface to Singular

2019.02.22

22 February 2019

**Marco Costantini**

**Willem Adriaan de Graaf**

**Willem Adriaan de Graaf**
Email: degraaf@science.unitn.it
Homepage: https://www.science.unitn.it/~degraaf/
Address: Willem de Graaf
Dipartimento di Matematica
Università degli Studi di Trento
I-38050 Povo (Trento)
Italy

# Contents

# Chapter 1

# singular: the GAP interface to Singular

## 1.1 Introduction

This is the manual of the GAP package "singular" that provides an interface from the GAP computer algebra system to the Singular computer algebra system.

This package allows the GAP user to access functions of Singular from within GAP, and to apply these functions to the GAP objects. With this package, the user keeps working with GAP and, if he needs a function of Singular that is not present in GAP, he can use this function via the interface; see the function `SingularInterface` (1.3.8).

This package provides also a function that computes Groebner bases of ideals in polynomial rings of GAP. This function uses the Singular implementation, which is very fast; see the function `GroebnerBasis` (1.5.1).

The interface is expected to work with every version of GAP 4, every (not very old) version of Singular, and on every platform, on which both GAP and Singular run; see paragraph 1.7.1 for details.

If you have used this package in the preparation of a paper please cite it as described in `https://www.gap-system.org/Contacts/cite.html`.

If GAP, Singular, and the GAP package singular are already installed and working on his computer, the user of this interface needs to read only the subsection `sing_exec` (1.2.4), the section 1.3, and in case of problems the subsection 1.7.4.

### 1.1.1 Package evolution

The work for the package singular has been started by Willem de Graaf, that planned this package as an interface to the function of Singular that calculates the Groebner bases. To this purpose, Willem de Graaf wrote the code for the conversion of rings and ideals from GAP to Singular, and the code for the conversion of numbers and polynomials in both directions.

Marco Costantini has widened the aim of the package, in order to make it a general interface to each possible function of Singular: with the function `SingularInterface` (1.3.8) it is possible to use from within GAP any function of Singular, including user-defined ones and future implementations. To this purpose, Marco Costantini has generalized the previous code for the conversion of objects in the new more general context, has written the code for the conversion of the various other types of objects, and has written the code for the low-level communication between GAP and Singular.

David Joyner has developed the code for the algebraic-geometric codes functions, and has written the corresponding section 1.6 of this manual.

Gema M. Diaz has helped with some testing and reports.

### 1.1.2 The system Singular

Singular is "A Computer Algebra System for Polynomial Computations" developed by G.-M. Greuel, G. Pfister, and H. Sch\"onemann, at Centre for Computer Algebra, University of Kaiserslautern. The authors of the GAP package singular are not involved in the development of the system Singular, and vice versa.

Singular is not included in this package, and can be obtained for free from `https://www.singular.uni-kl.de`. There, one can find also its documentation, installing instructions, the source code if wanted, and support if needed. Singular is available for several platforms.

A description of Singular, copied from its manual (paragraph "2.1 Background"), version 2-0-5, is the following:

" 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.). "

### 1.1.3 The system GAP

GAP stands for "Groups, Algorithms, and Programming", and is developed by several people ("The GAP Group").

GAP is not included in this package, and can be obtained for free from https://www.gap-system.org/. There, one can find also its documentation, installing instructions, the source code, and support if needed. The GAP system will run on any machine with an Unix-like or recent Windows or MacOS operating system and with a reasonable amount of ram and disk space.

A description of GAP, copied from its web site, is the following: "GAP is a system for computational discrete algebra, with particular emphasis on Computational Group Theory. GAP provides a programming language, a library of thousands of functions implementing algebraic algorithms written in the GAP language as well as large data libraries of algebraic objects. See the web site the overview and the description of the mathematical capabilities. GAP is used in research and teaching for studying groups and their representations, rings, vector spaces, algebras, combinatorial structures, and more. The system, including source, is distributed freely. You can study and easily modify or extend it for your special use."

## 1.2 Installation

In order to use this interface one must have both GAP version 4 and Singular installed.

### 1.2.1 Installing the system Singular

Follow the Singular installing instructions.

However, for a Unix system, one needs to download two files:

- `Singular-<version>-share.tar.gz`, that contains architecture independent data like documentation and libraries;
- `Singular-<version>-<uname>.tar.gz`, that contains architecture dependent executables, like the Singular program (precompiled). <uname> is a description of the processor and operating system for which Singular is compiled.

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.

Before trying the interface, make sure that Singular is installed and working as stand-alone program.

### 1.2.2 Installing the system GAP

Follow the GAP installing instructions.

However, the basic steps of a GAP installation are:

- Choose your preferred archive format and download the archives.
- Unpack the archives.
- On Unix: Compile GAP. (Compiled executables for Windows and Mac are in the archives.)
- On Unix: Some packages need further installation for full functionality (which is not available on Windows or Mac).

- Adjust some links/scripts/icons ..., depending on your system, to make the new version of GAP available to the users of your machine.
- Optional: Run a few tests.
- Optional, but appreciated: Give some feedback on your installation.

There is also an experimental Linux binary distribution via remote synchronization with a reference installation, which includes all packages and some optimizations. Furthermore, the Debian GNU/Linux distribution contains .deb-packages with the core part of GAP and some of the GAP packages.

### 1.2.3 Installing the package singular

The package singular is installed and loaded as a normal GAP package: see the GAP documentation (**Reference: Installing a GAP Package**) and (**Reference: Loading a GAP Package**).

Starting with version 4.4 of GAP, the package singular is distributed together with GAP. Hence, if GAP is already installed with all the distributed packages, then also the package singular is installed. However, if the package singular is not included in your GAP installation, it can be downloaded and unpacked in the `pkg/` directory of the GAP installation. If you don't have write access to the `pkg/` directory in your main GAP installation you can use private directories as explained in the GAP documentation (**Reference: GAP Root Directories**). The package singular doesn't require compilation.

### 1.2.4 sing_exec

▷ `sing_exec` (global variable)
▷ `sing_exec_options` (global variable)
▷ `SingularTempDirectory` (global variable)

In order to use the interface, GAP has to be told where to find Singular. This can be done in three ways. First, if the Singular executable file is in the search path, then GAP will find it. Second, it is possible to edit (before loading the package) one of the first lines of the file `singular/gap/singular.g` (that comes with this package). Third, it is possible to give the path of the Singular executable file directly during each GAP session assigning it to the variable *`sing_exec`* (after this package has been loaded, and before starting Singular), as in the example below.

Example

```
gap> LoadPackage( "singular" );
A GAP interface to Singular, by Marco Costantini and Willem de Graaf
true
gap> sing_exec:= "/home/wdg/Singular/2-0-3/ix86-Linux/Singular";;
```

The directory separator is always '/', even under DOS/Windows or MacOS. The value of *`sing_exec`* must refer to the text-only version of Singular (*`Singular`*), and not to the Emacs version (*`ESingular`*), nor to the terminal window version (*`TSingular`*).

In a similar way, it is possible to supply Singular with some command line options (or files to read containing user defined functions), assigning them to the variable *`sing_exec_options`*. This can be done by editing (before loading the package) one of the first lines of the file `singular/gap/singular.g` (that comes with this package), or directly during each GAP session (after this package has been loaded, and before starting Singular), as in the example below.

Example

```
gap> Add( sing_exec_options, "--no-rc" );
gap> Add( sing_exec_options, "/full_path/my_file" );
```

The variable *sing_exec_options* is initialized to *[ "-t" ]*; the user can add further options, but must keep *"-t"*, which is required. The possible options are described in the Singular documentation, paragraph "3.1.6 Command line options".

Singular is not executed in the current directory, but in a user-specified one, or in a temporary one. It is possible to supply this directory assigning it to the variable *SingularTempDirectory*. This can be done by editing (before loading the package) one of the first lines of the file `singular/gap/singular.g` (that comes with this package), or directly during each GAP session (after this package has been loaded, and before starting Singular), as in the example below. If *SingularTempDirectory* is not assigned, GAP will create and use a temporary directory, which will be removed when GAP quits.

Example

```
gap> SingularTempDirectory := Directory( "/tmp" );
dir("/tmp/")
```

On Windows, Singular version 3 may be not executed directly, but may be executed as *bash Singular*. In this case, the variables *sing_exec*, *sing_exec_options*, *SingularTempDirectory* must reflect this, otherwise Windows complains that *cygwin1.dll* is not found. The following works on my Windows machine.

Example

```
gap> LoadPackage("singular");
true
gap> SingularTempDirectory := Directory("c:/cygwin/bin");
dir("c:/cygwin/bin/")
gap> sing_exec := "c:/cygwin/bin/bash.exe";
"c:/cygwin/bin/bash.exe"
gap> sing_exec_options := [ "Singular", "-t" ];
[ "Singular", "-t" ]
gap> StartSingular();
```

Another possibility is to run Gap from within the Cygwin shell. In this case, with a standard installation of Cygwin and Singular, no change is required,

## 1.3 Interaction with Singular

The user must load the package singular with `LoadPackage` (**Reference: LoadPackage**).

### 1.3.1 StartSingular

▷ `StartSingular()` (function)

▷ `CloseSingular()` (function)

After the package singular has been loaded, Singular is started automatically when one of the functions of the interface is called. Alternatively, one can start Singular with the command *StartSingular*.

Example

```
gap> StartSingular();
```

See 1.7.1 for technical details. Explicit use of *StartSingular* is not necessary. If *StartSingular* is called when a previous Singular session is running, than session will be closed, and a new session will be started.

If at some point Singular is no longer needed, then it can be closed (in order to save system resources) with the command *CloseSingular*.

Example

```
gap> CloseSingular();
```

However, when GAP exits, it is expected to close Singular, and remove any temporary directory, except in the case of abnormal GAP termination.

### 1.3.2 SingularHelp

▷ SingularHelp(*topic*) (function)

Here *topic* is a string containing the name of a Singular topic. This function provides help on that topic using the Singular help system: see the Singular documentation, paragraphs "3.1.3 The online help system" and "5.1.43 help". If *topic* is the empty string "", then the title/index page of the manual is displayed.

This function can be used to display the Singular documentation referenced in this manual; *topic* must be given without the leading numbers.

Example

```
gap> SingularHelp( "" ); # a Mozilla window appears
#I  // ** Displaying help in browser 'mozilla'.
// ** Use 'system("--browser", <browser>);' to change browser,
// ** where <browser> can be: "mozilla", "xinfo", "info", "builtin", "dummy", \
"emacs".
```

The Singular function *system* can be accessed via the function SingularInterface (1.3.8). Some only-text browsers may be not supported by the interface.

### 1.3.3 Rings and orderings

All non-trivial algorithms in Singular require the prior definition of a (polynomial) ring, that will be called the "base-ring". Any polynomial (respectively vector) in Singular is ordered with respect to a term ordering (or, monomial ordering), that has to be specified together with the declaration of a ring. See the documentation of Singular, paragraph "3.3 Rings and orderings", for further information.

After defining in GAP a ring, a term ordering can be assigned to it using the function SetTermOrdering (1.3.5), and *after* the term ordering is assigned, the interface and Singular can be told to use this ring as the base-ring, with the function SingularSetBaseRing (1.3.6).

### 1.3.4 Supported coefficients fields

Let *p* be a prime, *pol* an irreducible polynomial, and *arg* an appropriate argument for the given function. The coefficient fields of the base-ring may be of the following form:

- *Rationals*,
- *CyclotomicField( arg )*,
- *AlgebraicExtension( Rationals, pol )*,
- *GaloisField( arg )* (both prime and non-prime),
- *AlgebraicExtension( GaloisField( p ), pol )*.

For some example see those for the function SetTermOrdering (1.3.5).

Let us remember that *CyclotomicField* and *GaloisField* can be abbreviated respectively to *CF* and *GF*; these forms are used also when GAP prints cyclotomic or Galois fields. See the GAP documentation about the functions: CyclotomicField (**Reference: CyclotomicField for (subfield and) conductor**), GaloisField (**Reference: GaloisField for field size**), AlgebraicExtension (**Reference: AlgebraicExtension**), and the chapters: (**Reference: Rational Numbers**), (**Reference: Abelian Number Fields**), (**Reference: Finite Fields**), (**Reference: Algebraic extensions of fields**).

### 1.3.5 SetTermOrdering

▷ SetTermOrdering(*R*) (function)
▷ TermOrdering(*R*) (attribute)

Let *R* be a polynomial ring. The value of *TermOrdering( R )* describes the term ordering of *R*, and can be a string, a list, or a monomial ordering of GAP. (The term orderings of Singular are explained in its documentation, paragraphs "3.3.3 Term orderings" and "B.2.1 Introduction to orderings".)

If this value is a string, for instance *"lp"* (lexicographical ordering), *"dp"* (degree reverse lexicographical ordering), or *"Dp"* (degree lexicographical ordering), this value will be passed to Singular without being interpreted or parsed by the interface.

If this value is a list, it must be of the form *[ str_1, d_1, str_2, d_2, ... ]*, where each *str_i* is a Singular ordering given as a string. Each *d_i* must be a number, and specifies the number of variables having that ordering; however, if *str_i* is a weighted order, like *"wp"* (weighted reverse lexicographical ordering) or *"Wp"* (weighted lexicographical ordering), then the corresponding *d_i* must be a list of positive integers that specifies the weight of each variable. The sum of the *d_i*'s (if numbers) or of their lengths (if lists) must be equal to the number of variables of the ring *R*.

This value can also be a monomial ordering of GAP: currently supported are *MonomialLexOrdering*, *MonomialGrevlexOrdering*, and *MonomialGrlexOrdering* (**Reference: Monomial Orderings**).

*TermOrdering* is a mutable attribute, see the GAP documentation of DeclareAttribute (**Reference: DeclareAttribute**); if it is changed on the GAP side, it is necessary thereafter to send again the ring to Singular with SingularSetBaseRing (1.3.6).

*SetTermOrdering* can be used to set the term ordering of a ring. It is not mandatory to assign a term ordering: if no term ordering is set, then the default *"dp"* will be used. If it is set, the term ordering must be set *before* the ring is sent to Singular with SingularSetBaseRing (1.3.6), otherwise, Singular will ignore that term ordering, and will use the previous value if any, or the default *"dp"*.

Example

```
gap> R1:= PolynomialRing( Rationals, ["x","y","z"] : old );;
gap> SetTermOrdering( R1, "lp" );
```

```
gap> R2:= PolynomialRing( GaloisField(9), 3 );;
gap> SetTermOrdering( R2, [ "wp", [1,1,2] ] );
gap> R3:= PolynomialRing( CyclotomicField(25), ["x","y","z"] : old );;
gap> SetTermOrdering( R3, MonomialLexOrdering() );
gap> x:=Indeterminate(Rationals);;
gap> F:=AlgebraicExtension(Rationals, x^5+4*x+1);;
gap> R4:= PolynomialRing( F, 6 );;
gap> SetTermOrdering( R4, [ "dp", 1, "wp", [1,1,2], "lp", 2 ] );
```

### 1.3.6 SingularSetBaseRing

▷ `SingularSetBaseRing(`*R*`)` (function)
▷ `SingularBaseRing` (global variable)

Here *R* is a polynomial ring. *SingularSetBaseRing* sets the base-ring in Singular equal to *R*. This ring will be also kept in GAP in the variable *SingularBaseRing*. After this assignment, all the functions of the interface will work with this ring. However, for some functions (those having rings, ideals, or modules as arguments) it is not necessary to explicitly set the base ring first, because in these cases the functions arguments contains information about a ring that will be used as a base-ring. This will be specified for each function in the corresponding section of this manual. (Unnecessary use of *SingularSetBaseRing* doesn't harm; forgetting to use *SingularSetBaseRing* produces the problem described in the paragraph 1.7.4.) The results of the computations may depend on the choice of the base-ring: see an example at `FactorsUsingSingular` (1.5.6), in which the factorization of $x^2+y^2$ is calculated.

Example

```
gap> R:= PolynomialRing( Rationals, ["x","y","z"] : old );;
gap> SingularSetBaseRing( R );
```

The value of *SingularBaseRing* when the package is loaded is *PolynomialRing( GF( 32003 ), 3 )*, in order to match the default base-ring of Singular.

### 1.3.7 SingularLibrary

▷ `SingularLibrary(`*string*`)` (function)

In Singular some functionality is provided by separate libraries that must be explicitly loaded in order to be used (see the Singular documentation, chapter "D. SINGULAR libraries"), see the example in `SingularInterface` (1.3.8).

The argument *string* is a string containing the name of a Singular library. This function makes sure that this library is loaded into Singular.

The functions provided by the library *ring.lib* could be not yet supported by the interface.

Example

```
gap> SingularLibrary( "general.lib" );
```

### 1.3.8 SingularInterface

▷ `SingularInterface(`*singcom*`, `*arguments*`, `*type_output*`)` (function)

The function *SingularInterface* provides the general interface that enables to apply the Singular functions to the GAP objects. Its arguments are the following:

- *singcom* is a Singular command or function (given as a string).
- *arguments* is a list of GAP objects, $O_1, O_2, ..., O_n$, that will be used as arguments of *singcom* (it may be the empty list). *arguments* may also be a string: in this case it is assumed that it contains one or more Singular identifiers, or a Singular valid expression, or something else meaningful for Singular, and it is passed to Singular without parsing or checking on the GAP side.
- *type_output* is the data type (given as a string) in Singular of the output. The data types are the following (see the Singular documentation, chapter "4. Data types"): "bigint", "def", "ideal", "int", "intmat", "intvec", "link", "list", "map", "matrix", "module", "number", "poly", "proc", "qring", "resolution", "ring", "string", "vector" (some of them were not available in previous versions of Singular). The empty string "" can be used if no output is expected. If in doubt you can use "def" (see the Singular documentation, paragraph "4.1 def"). Usually, in the documentation of each Singular function is given its output type.

Of course, the objects in the list *arguments* and the *type_output* must be appropriate for the function *singcom*: no check is done by the interface.

The function *SingularInterface* does the following:

1. converts each object $O_1, O_2, ..., O_n$ in *arguments* into the corresponding object $P_1, P_2, ..., P_n$, of Singular,
2. sends to Singular the command to calculate $singcom(P_1, P_2, ..., P_n)$,
3. gets the output (of type *type_output*) from Singular,
4. converts it to the corresponding Gap object, and returns it to the user.

The function *SingularInterface* is oriented towards the kind-of-objects/data-types, and not to the functions of Singular, because in this way it is much more general. The user can use "all" the existing functions of Singular and the interface is not bounded to the state of implementation of Singular: future functions and user-defined functions will be automatically supported.

The conversion of objects from Gap to Singular and from it back to Gap is done using some 'ad hoc' functions. Currently, the conversion of objects from GAP to Singular is implemented for the following types: "ideal", "int", "intmat", "intvec", "list", "matrix", "module", "number", "poly", "ring", "string", "vector". Objects of other types are not supported, or are even not yet implemented in GAP.

The conversion of objects from Singular to GAP is currently implemented for the following types: "bigint", "def", "ideal", "int", "intmat", "intvec", "list", "matrix", "module", "number", "poly", "proc" (experimental), "string", "vector". Objects of other types are returned as strings.

Before passing polynomials (or numbers, vectors, matrices, or lists of them) to Singular, it is necessary to have sent the base-ring to Singular with the function `SingularSetBaseRing` (1.3.6), in order to ensure that Singular knows about them. This is not necessary if in the input there is a ring, an ideal, or a module (before the polynomials), because these objects contain information about the ring to be used as base-ring. All the input must be relative to at most one ring; furthermore, at most one object of type "ring" can be in the input.

As SingularInterface is a rather general function, it is not guaranteed that it always works, and some functions are not supported. For instance, in Singular there is the function *pause* that waits until a keystroke is pressed; but the interface instead waits for the Singular prompt before sending it any new keystroke, and so calling *pause* would hang the interface. However, the unsupported functions like *pause* are only a few, and are not mathematically useful. SingularInterface tries to block calls to known unsupported functions.

Some Singular functions may return more than one value, see the Singular documentation, paragraph "6.2.7 Return type of procedures". In order to use one of these functions via *SingularInterface*, the type *type_output* must be "list". The output in GAP will be a list containing the values returned by the Singular function.

In the next example we compute the primary decomposition of an ideal. Note that for that we need to load the Singular library *primdec.lib*.

Example

```
gap> R:= PolynomialRing( Rationals, ["x","y","z"] : old );;
gap> i:= IndeterminatesOfPolynomialRing(R);;
gap> x:= i[1];; y:= i[2];; z:= i[3];;
gap> f:= (x*y-z)*(x*y*z+y^2*z+x^2*z);;
gap> g:= (x*y-z)*(x*y*z^2+x*y^2*z+x^2*y*z);;
gap> I:= Ideal( R, [f,g] );;
gap> SingularLibrary( "primdec.lib" );
gap> SingularInterface( "primdecGTZ", [ I ], "def" );
#I  Singular output of type "list"
[ [ <two-sided ideal in Rationals[x,y,z], (1 generators)>,
      <two-sided ideal in Rationals[x,y,z], (1 generators)> ],
  [ <two-sided ideal in Rationals[x,y,z], (1 generators)>,
      <two-sided ideal in Rationals[x,y,z], (1 generators)> ],
  [ <two-sided ideal in Rationals[x,y,z], (2 generators)>,
      <two-sided ideal in Rationals[x,y,z], (2 generators)> ],
  [ <two-sided ideal in Rationals[x,y,z], (3 generators)>,
      <two-sided ideal in Rationals[x,y,z], (2 generators)> ] ]
```

In the next example are calculated the first syzygy module of an ideal, and the resultant of two polynomials with respect a variable. Note that in this case it is not necessary to set the base-ring with SingularSetBaseRing (1.3.6), in the first case because the input *I* is of type "ideal", and in the second case because the base-ring was already sent to Singular in the former case.

Example

```
gap> R:= PolynomialRing( Rationals, ["x","y","z"] : old );;
gap> i:= IndeterminatesOfPolynomialRing( R );;
gap> x:= i[1];; y:= i[2];; z:= i[3];;
gap> f:= 3*(x+2)^3+y;;
gap> g:= x+y+z;;
gap> I:= Ideal( R, [f,g] );;
gap> M := SingularInterface( "syz", [ I ], "module" );;
gap> GeneratorsOfLeftOperatorAdditiveGroup( M );
[ [ -x-y-z, 3*x^3+18*x^2+36*x+y+24 ] ]
gap> SingularInterface( "resultant", [ f, g, z ], "poly");
3*x^3+18*x^2+36*x+y+24
```

### 1.3.9 SingularType

▷ `SingularType(obj)` (function)

to be written

## 1.4 Interaction with Singular at low level

### 1.4.1 SingularCommand

▷ `SingularCommand(precommand, command)` (function)

to be written

### 1.4.2 GapInterface

▷ `GapInterface(func, arg, out)` (function)

to be written

## 1.5 Other mathematical functions of the package

### 1.5.1 GroebnerBasis

▷ `GroebnerBasis(I)` (operation)

Here `I` is an ideal of a polynomial ring. This function computes a Groebner basis of `I` (that will be returned as a list of polynomials). For this function it is *not* necessary to set the base-ring with `SingularSetBaseRing` (1.3.6).

As term ordering, Singular will use the value of `TermOrdering` (1.3.5) of the polynomial ring containing `I`. Again, if this value is not set, then the degree reverse lexicographical ordering (`"dp"`) will be used.

Example

```
gap> R:= PolynomialRing( Rationals, ["x","y","z"] : old );;
gap> x := R.1;; y := R.2;; z := R.3;;
gap> r:= [ x*y*z -x^2*z, x^2*y*z-x*y^2*z-x*y*z^2, x*y-x*z-y*z ];;
gap> I:= Ideal( R, r );
<two-sided ideal in Rationals[x,y,z], (3 generators)>
gap> GroebnerBasis( I );
[ x*y-x*z-y*z, x^2*z-x*z^2-y*z^2, x*z^3+y*z^3, -x*z^3+y^2*z^2-y*z^3 ]
```

### 1.5.2 SINGULARGBASIS

▷ `SINGULARGBASIS` (global variable)

This variable is a record containing the component `GroebnerBasis`. When the variable SINGULARGBASIS is assigned to the GAP global variable `GBASIS`, then the computations of Groebner

bases via GAP's internal function for that, `GroebnerBasis` (**Reference: GroebnerBasis**), are done by Singular.

Singular claims that it "features one of the fastest and most general implementations of various algorithms for computing Groebner bases". The GAP's internal function claims to be "a na{\"\i}ve implementation of Buchberger's algorithm (which is mainly intended as a teaching tool): it might not be sufficient for serious problems."

(Note in the following example that the Groebner bases calculated by the GAP internal function are in general not reduced; for reduced bases see the GAP function `ReducedGroebnerBasis` (**Reference: ReducedGroebnerBasis**).)

Example

```
gap> R:= PolynomialRing( Rationals, 3 );;
gap> i:= IndeterminatesOfPolynomialRing( R );;
gap> pols:= [i[1]+i[2]+i[3], i[1]*i[2]+i[1]*i[3]+i[2]*i[3], i[1]*i[2]*i[3]];;
gap> o:= MonomialLexOrdering();;
gap> GBASIS:= GAPGBASIS;;
gap> GroebnerBasis( pols, o ); # This is the internal GAP method.
[ x+y+z, x*y+x*z+y*z, x*y*z, -y^2-y*z-z^2, z^3 ]
gap> GBASIS:= SINGULARGBASIS;;
gap> GroebnerBasis( pols, o ); # This uses Singular via the interface.
[ z^3, y^2+y*z+z^2, x+y+z ]
```

### 1.5.3 HasTrivialGroebnerBasis

▷ `HasTrivialGroebnerBasis(`*I*`)` (function)

The function *HasTrivialGroebnerBasis* returns *true* if the Groebner basis of the ideal *I* is trivial, false otherwise. This function can be used if it is not necessary to know the Groebner basis of an ideal, but it suffices to know only whether it is trivial or not.

Example

```
gap> x:= Indeterminate( Rationals, "x" : old );;
gap> y:= Indeterminate( Rationals, "y", [ x ] : old );;
gap> z:= Indeterminate( Rationals, "z", [ x, y ] : old );;
gap> R:= PolynomialRing( Rationals, [ x, y, z] );;
gap> f:= (x*y-z)*(x*y*z+y^2*z+x^2*z);;
gap> g:= (x*y-z)*(x*y*z^2+x*y^2*z+x^2*y*z);;
gap> I:= Ideal( R, [f,g] );;
gap> HasTrivialGroebnerBasis( I );
false
```

### 1.5.4 GcdUsingSingular (for polynomials)

▷ `GcdUsingSingular(`*pol_1, pol_2, ..., pol_n*`)` (function)
▷ `GcdUsingSingular(`*[pol_1, pol_2, ..., pol_n]*`)` (function)

The arguments of this function are (possibly multivariate) polynomials separated by commas, or it is a list of polynomials. This function returns the greatest common divisor of these polynomials. For this function it is *necessary* for the polynomials to lie in the base-ring, as set by `SingularSetBaseRing` (1.3.6).

Example

```
gap> R:= PolynomialRing( Rationals, ["x","y","z"] : old );;
gap> SingularSetBaseRing( R );
gap> i:= IndeterminatesOfPolynomialRing(R);;
gap> x:= i[1];; y:= i[2];; z:= i[3];;
gap> f:= (x*y-z)*(x*y*z+y^2*z+x^2*z);
x^3*y*z+x^2*y^2*z+x*y^3*z-x^2*z^2-x*y*z^2-y^2*z^2
gap> g:= (x*y-z)*(x*y*z^2+x*y^2*z+x^2*y*z);
x^3*y^2*z+x^2*y^3*z+x^2*y^2*z^2-x^2*y*z^2-x*y^2*z^2-x*y*z^3
gap> GcdUsingSingular( f, g );
-x*y*z+z^2
```

### 1.5.5 FactorsUsingSingularNC

▷ `FactorsUsingSingularNC(`*f*`)` (function)

Here *f* is a (possibly multivariate) polynomial. This function returns the factorization of *f* into irreducible factors. The first element in the output is a constant coefficient, and the others may be monic (with respect to the term ordering) polynomials, as returned by Singular. For this function it is *necessary* that *f* lies in the base-ring, as set by `SingularSetBaseRing` (1.3.6).

The function does not check that the product of these factors gives *f* (for that use `FactorsUsingSingular` (1.5.6)): Singular version 2-0-3 contains a bug so that the Singular function *factorize* may give wrong results (therefore Singular version at least 2-0-4 is recommended).

Example

```
gap> R:= PolynomialRing( Rationals, ["x","y","z"] : old );;
gap> SingularSetBaseRing( R );
gap> i:= IndeterminatesOfPolynomialRing( R );;
gap> x:= i[1];; y:= i[2];; z:= i[3];;
gap> f:= (x*y-z)*(3*x*y*z+4*y^2*z+5*x^2*z);
5*x^3*y*z+3*x^2*y^2*z+4*x*y^3*z-5*x^2*z^2-3*x*y*z^2-4*y^2*z^2
gap> FactorsUsingSingularNC( f );
[ 1, -5*x^2-3*x*y-4*y^2, -x*y+z, z ]
gap> f:= (x*y-z)*(5/3*x*y*z+4*y^2*z+6*x^2*z);
6*x^3*y*z+5/3*x^2*y^2*z+4*x*y^3*z-6*x^2*z^2-5/3*x*y*z^2-4*y^2*z^2
gap> FactorsUsingSingularNC( f );
[ 1/3, -18*x^2-5*x*y-12*y^2, -x*y+z, z ]
```

### 1.5.6 FactorsUsingSingular

▷ `FactorsUsingSingular(`*f*`)` (function)

This does the same as `FactorsUsingSingularNC` (1.5.5), except that on the GAP level it is checked that the product of these factors gives *f*. Again it is *necessary* that *f* lies in the base-ring, as set by `SingularSetBaseRing` (1.3.6).

Example

```
gap> R:= PolynomialRing( Rationals, ["x","y"] : old );;
gap> SingularSetBaseRing( R );
gap> x := R.1;; y := R.2;;
gap> FactorsUsingSingular( x^2 + y^2 );
[ 1, x^2+y^2 ]
```

```
gap> R:= PolynomialRing( GaussianRationals, ["x","y"] : old);;
gap> SingularSetBaseRing( R );
gap> x := R.1;; y := R.2;;
gap> FactorsUsingSingular( x^2 + y^2 );
[ 1, x+E(4)*y, x-E(4)*y ]
```

### 1.5.7 GeneratorsOfInvariantRing

▷ `GeneratorsOfInvariantRing(R, G)` (function)

Here $R$ is a polynomial ring, and $G$ a finite group, which is either a matrix group or a permutation group. If $G$ is a matrix group, then its degree must be less than or equal to the number of indeterminates of $R$. If $G$ is a permutation group, then its maximal moved point must be less than or equal to the number of indeterminates of $R$. This function computes a list of generators of the invariant ring of $G$, corresponding to its action on $R$. This action is taken to be from the left.

For this function it is *not* necessary to set the base-ring with `SingularSetBaseRing` (1.3.6).

Example

```
gap> m:=[[1,1,1],[0,1,1],[0,0,1]] * One( GF(3) );;
gap> G:= Group( [m] );;
gap> R:= PolynomialRing( GF(3), 3 );;
gap> GeneratorsOfInvariantRing( R, G );
[ x_3, x_1*x_3+x_2^2+x_2*x_3, x_1^3+x_1^2*x_3-x_1*x_2^2-x_1*x_2*x_3 ]
```

## 1.6 Algebraic-geometric codes functions

This section of GAP's singular package and the corresponding code were written by David Joyner, wdj@usna.edu, (with help from Christoph Lossen and Marco Costantini). It has been tested with Singular versrion 2.0.x.

To start off, several new Singular commands must be loaded. The following command loads the necessary Singular and GAP commands, the packages singular and GUAVA (if not already loaded), and (re)starts Singular.

Example

```
gap> ReadPackage("singular", "contrib/agcode.g");;
```

### 1.6.1 AllPointsOnCurve

▷ `AllPointsOnCurve(f, F)` (function)

Let $F$ be a finite and prime field. The function *`AllPointsOnCurve( f, F )`* computes a list of generators of maximal ideals representing rationals points on a curve $X$ defined by $f(x,y) = 0$.

Example

```
gap> F:=GF(7);;
gap> R2:= PolynomialRing( F, 2 );;
gap> SetTermOrdering( R2, "lp" );; # --- the term ordering must be "lp"
gap> indet:= IndeterminatesOfPolynomialRing(R2);;
gap> x:= indet[1];; y:= indet[2];;
gap> f:=x^7-y^2-x;;
gap> AllPointsOnCurve(f,F);
```

```
[ [ x_1 ], [ x_1-Z(7)^0 ], [ x_1+Z(7)^4 ], [ x_1+Z(7)^5 ], [ x_1+Z(7)^0 ],
  [ x_1+Z(7) ], [ x_1+Z(7)^2 ] ]
```

### 1.6.2 AGCode

▷ AGCode(*f, G, D*) (function)

Let f be a polynomial in x,y over F=GF(p) representing plane curve $X$ defined by $f(x,y) = 0$, where p is a prime (prime powers are not yet supported by the underlying Singular function). Let G, D be disjoint rational divisors on $X$, where D is a sum of distinct points, $supp(D) = P_1, ..., P_n$. The AG code associated to f, G, D is the F defined to be the image of the evaluation map $f \mapsto (f(P_1), ..., f(P_n))$. The function *AGCode* computes a list of length three, [G, n, k], where G is a generator matrix of the AG code C, n is its length, and k is its dimension.

Example

```
gap> F:=GF(7);;
gap> R2:= PolynomialRing( F, 2 );;
gap> SetTermOrdering( R2, "lp" );; # --- the term ordering must be "lp"
gap> SingularSetBaseRing(R2);
gap> indet:= IndeterminatesOfPolynomialRing(R2);;
gap> x:= indet[1];; y:= indet[2];;
gap> f:=x^7-y^2-x;;
gap> G:=[2,2,0,0,0,0,0]; D:=[4..8];
[ 2, 2, 0, 0, 0, 0, 0 ]
[ 4 .. 8 ]
gap> agc:=AGCode(f,G,D);
[ [ [ Z(7)^3, Z(7), 0*Z(7), Z(7)^4, Z(7)^5 ],
      [ 0*Z(7), Z(7)^4, Z(7)^0, Z(7)^5, Z(7)^3 ],
      [ 0*Z(7), 0*Z(7), Z(7)^3, Z(7), Z(7)^2 ] ], 5, 3 ]
```

This generator matrix can be fed into the GUAVA command `GeneratorMatCode` (**GUAVA: GeneratorMatCode**) to create a linear code in GAP, which in turn can be fed into the GUAVA command `MinimumDistance` (**GUAVA: MinimumDistance**) to compute the minimum distance of the code.

Example

```
gap> ag_mat:=agc[1];;
gap> C := GeneratorMatCode( ag_mat, GF(7) );
a linear [5,3,1..3]2 code defined by generator matrix over GF(7)
gap> MinimumDistance(C);
3
```

## 1.7 Troubleshooting and technical stuff

### 1.7.1 Supported platforms and underlying GAP functions

This package has been developed mainly on a Linux platform, with GAP version 4.4, and Singular version 2-0-4. A reasonable work has been done to ensure backward compatibility with previous versions of GAP 4, but some features may be missing. This package has been tested also with some other versions of Singular, including 2-0-3, 2-0-5, and 2-0-6, and on other Unix systems. It has been tested also on Windows, but it is reported to be slower that on Linux.

There is an extension of Singular, named Plural, which deals with certain noncommutative polynomial rings; see the Singular documentation, section "7. PLURAL". Currently, GAP doesn't support these noncommutative polynomial rings. The user of the Singular may use the features of Plural by calling the Singular function *ncalgebra* via *SingularInterface*. In this case, extreme care is needed, because on the GAP side the polynomial will still be commutative.

For the low-level communication with Singular, the interface relies on the GAP function InputOutputLocalProcess (**Reference: InputOutputLocalProcess**), and this function is available only in GAP 4.2 (or newer) on a Unix environment or in GAP 4.4 (or newer) on Windows; auto-detection is used. In this case, GAP interacts with a unique continuous session of Singular.

In the case that the GAP function *InputOutputLocalProcess* is not available, then the singular interface will use the GAP function Process (**Reference: Process**). In this case only a limited subset of the functionality of the interface are available: for example StartSingular (1.3.1) and GeneratorsOfInvariantRing (1.5.7) are not available, but GroebnerBasis (1.5.1) is; SingularInterface (1.3.8) supports less data types. In this case, for each function call, a new session of Singular is started and quitted.

### 1.7.2 How different versions of GAP display polynomial rings and polynomials

The way in which GAP displays polynomials has changed passing from version 4.3 to 4.4 and the way in which GAP displays polynomial rings has changed passing from version 4.4 to 4.5.

Example

```
gap> # GAP 4.3 or older
gap> R := PolynomialRing( Rationals, [ "x" ] : new );
PolynomialRing(..., [ x ])
gap> x := IndeterminatesOfPolynomialRing( R )[1];;
gap> x^2 + x;
x+x^2
```

Example

```
gap> # GAP 4.4
gap> R := PolynomialRing( Rationals, [ "x" ] : new );
PolynomialRing(..., [ x ])
gap> x := IndeterminatesOfPolynomialRing( R )[1];;
gap> x^2 + x;
x^2+x
```

Example

```
gap> # GAP 4.5 or newer
gap> R := PolynomialRing( Rationals, [ "x" ] : new );
Rationals[x]
gap> x := IndeterminatesOfPolynomialRing( R )[1];;
gap> x^2 + x;
x^2+x
```

The examples in this manual use the way of displaying of the newest GAP.

### 1.7.3 Test file

The following performs a test of the package functionality using a test file (**Reference: Test Files**).

Example

```
gap> fn := Filename( DirectoriesPackageLibrary( "singular", "tst" ), "testall.tst" );;
gap> Test( fn );
true
```

### 1.7.4 Common problems

A common error is forgetting to use SingularSetBaseRing (1.3.6). In the next example, *SingularInterface* works only after having used *SingularSetBaseRing*.

Example

```
gap> a:=Indeterminate( Rationals );;
gap> F:=AlgebraicExtension( Rationals, a^5+4*a+1 );;
gap> R:=PolynomialRing( F, ["x","y"] : old );;
gap> x := R.1;; y := R.2;;
gap> SingularInterface( "lead", [x^3*y+x*y+y^2], "poly" );
Error, sorry: Singular, or the interface to Singular, or the current
SingularBaseRing, do not support the object x^3*y+x*y+y^2.
Did you remember to use 'SingularSetBaseRing' ?
[...]
brk> quit;
gap> SingularSetBaseRing( R );
gap> SingularInterface( "lead", [x^3*y+x*y+y^2], "poly" );
x^3*y
```

A corresponding problem would happen if the user works directly with Singular and forgets to define the base-ring at first.

As explained in the GAP documentation (**Reference: Polynomials and Rational Functions**), given a ring *R*, GAP does not consider *R* as a subset of a polynomial ring over *R*: for example the zero of *R* (0) and the zero of the polynomial ring ($0x^0$) are different objects. GAP prints these different objects in the same way, and this fact may be misleading. This is a feature of GAP independent from the package singular, but it is important to keep it in mind, as most of the objects used by Singular are polynomials, or their coefficients.

### 1.7.5 Errors on the Singular side

Errors may occur on the Singular side, for instance using SingularInterface (1.3.8) if the arguments supplied are not appropriate for the called function. In general, it is still an open problem to find a satisfactory way to handle in GAP the errors of this kind.

At the moment, when an error on the Singular side happens, Singular may print an error message on the so-called "standard error"; this message may appear on the screen, but it is not logged by the GAP function LogTo (**Reference: LogTo**). The interface prints *No output from Singular*, and then the trivial object (of the type specified as the third argument of *SingularInterface*) may be returned.

### 1.7.6 Sending a report

As every software, also this package may contain bugs. If you find a bug, or a missing feature, or some other problem, or if you have comments and suggestions, or if you need some help, you may do so

via our issue tracker at https://github.com/gap-packages/singular/issues. Please include in the report the code that causes the problem, so that we can replicate the problem.

If appropriate, you can set InfoSingular (1.7.8) to 3, to see what happens between GAP and Singular (but this may give a lot of output). Note that LogTo (**Reference: LogTo**) does not log messages written directly on the screen by Singular.

Every report about this package is welcome, however the probability that your problem will be fixed quickly increases if you read the text "How to Report Bugs Effectively", https://www.chiark.greenend.org.uk/~sgtatham/bugs.html , and send a bug report according to this text.

### 1.7.7 SingularReportInformation

▷ SingularReportInformation() (function)

The function *SingularReportInformation* collects a description of the system, which should be included in any bug report.

Example

```
gap> SingularReportInformation();
Pkg_Version := "4.04.15";
Gap_Version := "4.dev";
Gap_Architecture := "i686-pc-linux-gnu-gcc";
Gap_BytesPerVariable := 4;
uname := "Linux 2.4.20 i686";
Singular_Version: := 2004;
Singular_Name: := "/usr/local/Singular/2-0-4/ix86-Linux/Singular";

"Pkg_Version := \"4.04.15\";\nGap_Version := \"4.dev\";\nGap_Architecture := \
\"i686-pc-linux-gnu-gcc\";\nGap_BytesPerVariable := 4;\nuname := \"Linux 2.4.2\
0 i686\";\nSingular_Version: := 2004;\nSingular_Name: := \"/usr/local/Singular\
/2-0-4/ix86-Linux/Singular\";\n"
```

### 1.7.8 InfoSingular

▷ InfoSingular (info class)

This is the info class (**Reference: Info Functions**) used by the interface. It can be set to levels 0, 1, 2, and 3. At level 0 no information is printed on the screen. At level 1 (default) the interface prints a message about the *type_output*, when "def" is used in *SingularInterface*, see the example at SingularInterface (1.3.8). At level 2, information on the activities of the interface is printed (e.g., messages when a Singular session, or a Groebner basis calculation, is started or terminated). At level 3 all strings that GAP sends to Singular are printed, as well as all strings that Singular sends back.

Example

```
gap> SetInfoLevel( InfoSingular, 2 );
gap> G:= SymmetricGroup( 3 );;
gap> R:= PolynomialRing( GF(2), 3 );;
gap> GeneratorsOfInvariantRing( R, G );
#I  running SingularInterface( "invariant_ring", [ "matrix", "matrix"
 ], "list" )...
#I  done SingularInterface.
```

```
[ x_1+x_2+x_3, x_1*x_2+x_1*x_3+x_2*x_3, x_1*x_2*x_3 ]
gap> I:= Ideal( R, last );;
gap> GroebnerBasis( I );
#I  running GroebnerBasis...
#I  done GroebnerBasis.
[ x_1+x_2+x_3, x_2^2+x_2*x_3+x_3^2, x_3^3 ]
gap> SetInfoLevel( InfoSingular, 1 );
```

# Index