SINGULAR Quick Reference

SINGULAR Version 3-0

Do not forget to terminate all commands with a `;` (semicolon)!
In particular if SINGULAR prints the continuation prompt `.` (period) instead of the regular command prompt `>`, then it waits for a command to be terminated by a `;`. If that does not help, try one or more `"` or `}` to close an opened string or block.
Comments start with `//` and extend to end of line.

Some of the topics concerning interactive use are system dependent.

Starting SINGULAR

`Singular`	start SINGULAR
`Singular` *file* ...	read *files* and prompt for further commands
`Singular --help`	print help on command line options and exit

Stopping SINGULAR

`quit;`	exit SINGULAR; also `exit;` or `$`
`C-c`	interrupt SINGULAR

Getting help

`help;`	enter online help system
`help` *topic*`;`	describe *topic*; also `?` *topic*`;`

Inside the info help system:

`C-h`	get help on help system
`q`	exit from help system
`n/p/u`	go to next/previous/upper node
`m`	pick menu item by name
`l`	go to last visited node/exit from help on help
`SPC/DEL`	scroll forward/backward one page

Commandline editing

Commandline editing is similar to that of, e.g., **bash** or **tcsh**:

`BS/C-d`	remove character on the left/right of cursor
`C-p/C-n`	get previous/next line from history
`C-b/C-f`	move cursor left/right
`C-a/C-e`	go to beginning/end of line
`C-u/C-k`	delete to beginning/end of line

Names and objects

Names (= identifiers) have to be declared before they are used:

type name [= *expression*]`;`
declare variable *name*

`kill(`*name*`)` delete variable *name*

Names of type `number`, `poly`, `ideal`, `vector`, `module`, `matrix`, `map`, and `resolution` may be declared only inside a ring. They are local to that ring. The same holds for a `list` if it contains an object of the above types. All other types may be declared at any time. They are globally visible.

Names may consist of alphanumeric characters including `_` (underscore) and have to start with a letter. Capital and small letters are distinguished. Names may be followed by an integer expression in parentheses, resulting in so-called *indexed names*.

name`(`*n*`..`*m*`)`	shortcut for *name*`(`*n*`)`, ..., *name*`(`*m*`)` (e.g. `ring r = 0, x(1..3), dp;`)
`_` (underscore)	refers to the value of the last expression printed

Ring declaration

`ring` *name* `=` *basefield*`, (`*ringvars*`),` *ordering*`;`
declare ring *name* and make it the new basering. *ringvars* has to be a list of names, the other items are described below. Example:
`ring r = 32003, (x, y, z), dp;`

`qring` *name* `=` *ideal*`;`
declare quotient ring *name* of the current basering with respect to *ideal*. *ideal* has to be a standard basis. Make *name* the new basering.

Examples of available *basefields*:

`0`	the rational numbers
p	the finite field Z_p with p elements, $2 \leq p \leq 2147483629$ a prime
`(`*p*`^`*n*`,` *gen*`)`	the finite field with p^n elements, p a prime and $4 \leq p^n \leq 32671$. The name *gen* refers to some generator of the cyclic group of unities.
`(`*p*`,` *alpha*`)`	algebraic extension of Q or Z_p ($p = 0$ or as above) by *alpha*. The minpoly μ_{alpha} for *alpha* has to be specified with an assignment to `minpoly` (e.g. `minpoly=a^2+1;`, for *alpha* = `a`). *alpha* has to be a name.
`(`*p*`,` t_1`, ...)`	transcendental extension of Q or Z_p ($p = 0$ or as above) by t_i. The t_i have to be names.
`real,`*len*	the real numbers represented by long floating point numbers of length*len*

Term orderings

An *ordering* as referred to in the ring declaration may either be a global, local, or matrix ordering or a list of these resulting in a product ordering. The list may include extra weight vectors and may be preceded or followed by a module ordering specification.

Global orderings

`lp`	lexicographical ordering
`dp`	degree reverse lexicographical ordering
`Dp`	degree lexicographical ordering
`wp(`w_1`, ...)`	weighted reverse lexicographical ordering
`Wp(`w_1`, ...)`	weighted lexicographical ordering. The w_i have to be positive integers.

Local orderings

`ls`	negative lexicographical ordering
`ds`	negative degree reverse lexicographical ordering
`Ds`	negative degree lexicographical ordering
`ws(`w_1`, ...)`	general weighted reverse lexicographical ordering
`Ws(`w_1`, ...)`	general weighted lexicographical ordering. w_1 has to be a non-zero integer, every other w_i may be any integer

Matrix orderings

`M(`m_{11}`,` m_{12}`, ...,` m_{nn}`)`
m has to be an invertible matrix with integer coefficients. Coefficients have to be specified row-wise.

Product orderings

`(`o_1`[(`k_1`)],` o_2`[(`k_2`)], ...,` o_n`[(`k_n`)])`
the o_i have to be any of the above orderings. `lp`, `dp`, `Dp`, `ls`, `ds`, `Ds` may be followed by an integer expression k_i in parentheses specifying the number of variables o_i refers to (e.g. `(lp(3), dp(2))`).

Extra weight vector

`a(`w_1`, ...)`	any of the above degree orderings may be preceded by an extra weight vector

Module orderings

`(`*c*`,` o_1`, ...)`	sort by components first
`(`o_1`, ...,` *c*`)`	sort by variables first. o_i may be any of the above orderings or an extra weight vector, *c* may be one of `C` or `c`:
`C`	sort generators in ascending order (i.e. `gen(`i`)` $<$ `gen(`j`)` iff $i < j$)
`c`	sort generators in descending order

Data types

Examples of ring-independent types:

```
int i1 = 101; int i2 = 13 div 3;
intvec iv = 13 div 3, -4, i1;
intmat im[2][2] = 13 div 3, -4, i1;
```

a 2×2 matrix. Entries are filled row-wise, missing entries are set to zero, extra entries are ignored. vector/matrix elements are accessed using the `[...]` operator, where the first element has index one (e.g. `iv[3]; im[1, 2];`).

```
string s1 = "a quote \" and a backslash \\";
string s2 = "con" + "catenation";
```

Basering in the following is `ring r = 0, (x, y, z, mu, nu), dp;`

```
number n = 5/3;
poly p(1) = 3/4x3yz4+2xy2;
poly p(2) = (5/3)*mu^2*nu^3+n*yz2;
```

`p(1)` equals $3/4x^3yz^4 + 2xy^2$. Short format of monomials is valid for one-character ring variables only.

`ideal i = p(1..2), x+y;`
note the use of indexed names

```
vector v = [p(1), p(2), x+y];
vector w = 2*p(1)*gen(6)+n*nu*gen(1);
```

vectors may be written in brackets (`[...]`) or expressed as linear combinations of the canonical generators `gen(`i`)`

```
module mo = v, w, x+y*gen(1);
resolution r = sres(std(mo), 0);
matrix ma[2][2] = 5/3, p(1), 101;
```

the rules for declaring, filling, and accessing integer matrices apply to types `matrix` and `vector`, too

`list l = iv, v, p(1..2), mo;`
lists may collect objects of any type. They are ring-dependent iff one of the entries is.

`def d = read("MPfile:r example.mp");`
a name of type `def` inherits the type of the object assigned first to it. Useful if the actual type of an object is unknown.

Monitoring and debugging tools

`timer = 1;`	print time used for commands to execute
`int t = timer;` *command*`; ...; timer-t;`	print time used for *commands* to execute
`memory(1);`	print number of bytes allocated from system
`option(prot);`	show algorithm protocol
`option(mem);`	show algorithm memory usage
`TRACE = 1;`	print protocol on execution of procedures
`listvar(all);`	list all (user-)defined names
`listvar(`*ringname*`);`	list all names belonging to *ringname*

Options

`option();` show current option settings

`option(`*option*$_1$`, no`*option*$_2$`, ...);`
switch *option*$_1$ on and *option*$_2$ off, resp.

`option(none);` reset all options to default values

Type `help option;` for a list of all options.

Monitoring

`debugLib` show loading of procedures from libraries
`mem` show algorithm memory usage
`prot` show algorithm protocol

Standard bases

`fastHC` try to find highest corner as fast as possible
`intStrategy` avoid divisions
`morePairs` create additional pairs
`notSugar` disable sugar strategy
`redSB` compute reduced standard bases
`redTail` reduce tails
`sugarCrit` use sugar criteria
`weightM` automatically compute weights

Resolutions

`minRes` do additional minimizing
`notRegularity` disable regularity bound

Miscellany

`returnSB` let some functions return standard bases

System variables

Type `help System variables;` for a list of all system variables.

Standard bases

`degBound` stop if (weighted) total degree exceeds `degBound`
`multBound` stop if multiplicity gets smaller than `multBound`
`noether` cut off all monomials above monomial `noether`

Miscellany

`basering` current basering
`minpoly` minimal polynomial for algebraic extensions
`short` do not print monomials in short format if zero
`timer` on assignment of a non-zero value show time used for execution of executed commands. On evaluation, return system time in seconds used by SINGULAR since start
`TRACE` print information on procedures being executed if larger than one

Input and output

`< "`*filename*`";` load and execute *filename*

`write("`*filename*`", `*expression*`, ...)`
write *expressions* to ASCII file *filename*

`read("`*filename*`");`
read ASCII file *filename* and return content as a string. See also example below.

`dump("MPfile: `*filename*`");`
`getdump("MPfile: `*filename*`");`
dump current state of SINGULAR to *filename* and retrieve it, resp.

An example how to write one single expression (in this case the ideal `i`) to a file and read it back from there:

```
write("i.save", i);
execute("ideal i=" + read("i.save") + ";");
```

Libraries

`LIB "`*library*`";` load *library*
`help `*library*`;` show help on *library*
`help all.lib;` show list of all libraries

Mapping

`map `*name*` = `*ringname*`, `*ideal*`;`
declare a map *name* from *ringname* to current basering. The i-th ring variable from *ringname* is mapped to the i-th generator of *ideal*.

mapname`(`*expression*`)`
apply map *mapname* to *expression*

Coefficients between rings with different basefields are mapped in the following way (non-canonical maps only):

$$Z_p \to Q : [i]_p \mapsto i \in [-p/2, p/2] \subset Z$$
$$Z_p \to Z_q : [i]_p \mapsto i \in [-p/2, p/2] \subset Z, i \mapsto [i]_q$$

`fetch(`*ringname*`, `*name*`)`
map from ring *ringname* to current basering. The rings have to be identical up to names of ring variables

`imap(`*ringname*`, `*name*`)`
map from subring *ringname* to current basering

`subst(`*expression*`, `*ringvar*`, `*monomial*`)`
substitute *ringvar* by *monomial* in *expression*

Miscellany

`setring(`*ringname*`)`
make *ringname* the current basering

Data on polynomials

`ord(`*poly* | *vector*`)`
return (weighted) degree of initial term

`deg(`*poly* | *vector*`)`
return maximal (weighted) degree

`size(`*ideal* | *module*`)`
`size(`*poly* | *vector*`)`
`size(`*string* | *intvec* | *list*`)`
return (1) number of non-zero generators; (2) number of monomials; (3) length

`lead(`*expression*`)`
return initial term(s)

Operations on polynomials

`gcd(`*poly*$_1$`, `*poly*$_2$`)`
return greatest common divisor

`factorize(`*poly*`[, `*int*`])`
return irreducible factors. Return constant factor and multiplicities in dependency on *int*.

Differentiation and jets

`diff(`*expression*`, `*ringvar*`)`
`diff(`*ideal*$_1$`, `*ideal*$_2$`)`
(1) return partial derivation by *ringvar*; (2) differentiate each elt. of *ideal*$_2$ by the differential operators corresponding to the elements of *ideal*$_1$

`jacob(`*poly* | *ideal*`)`
return jacobi ideal or matrix, resp.

`jet(`*expression*`, `*int*`[, `*intvec*`])`
return *int*-jet of *expression*. Return weighted *int*-jet if *intvec* is specifified.

Standard bases

`groebner(`*ideal* | *module*`[, `*int*`])`
compute a standard basis (SB) of *ideal* resp. *module* using a heuristically chosen method. Delimit computation time to *int* seconds.

`std(`*ideal* | *module*`[, `*intvec*`])`
compute a SB. Use first Hilbert series *intvec* (result from `hilb(..., 1)`) for Hilbert-driven computation.

`stdfglm(`*ideal*`[, `*string*`])`
use FGLM algorithm to compute a SB from a SB w.r.t. the "simpler" ordering *string* (defaults to `dp`)

`stdhilb(`*ideal*`[, `*intvec*`])`
use Hilbert-driven algorithm to compute a SB. If Hilbert series *intvec* is not specified compute it first.

`fglm(`*ringname*`, `*idealname*`)`
use FGLM algorithm to transform SB *idealname* from ring *ringname* to a SB w.r.t. the ordering of the current basering

`reduce(`*expression*`, `*ideal* | *module*`[, `*int*`])`
reduce *expression* w.r.t. second argument which should be a SB. Use lazy reduction if *int* equals one.

Computation of invariants

Most of the results are meaningful only if the input ideal or module is represented by a standard basis.

`degree(`*ideal* | *module*`)`
display (Krull) dimension, codimension and multiplicity

`dim(`*ideal* | *module*`)`
return (Krull) dimension

`hilb(`*ideal* | *module*`[, `*int*`])`
display first and second Hilbert series with one argument. Return *int*-th Hilber series otherwise ($int = 1, 2$).

`mult(`*ideal* | *module*`)`
return multiplicity

`vdim(`*ideal* | *module*`)`
return vector space dimension of current basering modulo *ideal* or *module*, resp.

Resolutions

An integer argument *length* in the following descriptions specifies the length of the resolution to compute. If *length* equals zero, the whole resolution is computed.

`res(`*ideal* | *module*`, `*length*`[, `*int*`])`
compute a free resolution (FR) of *ideal* resp. *module* using a heuristically chosen method. Compute a minimal resolution if a third argument is given.

`mres(`*ideal* | *module*`, `*length*`)`
compute a minimal FR using the standard basis method

`lres(`*ideal* | *module*`, `*length*`)`
compute a FR using LaSacala's method

`sres(`*ideal* | *module*`, `*length*`)`
compute a FR using Schreyer's method

`syz(`*ideal* | *module*`)`
compute the first syzygy

`minres(`*resolution* | *list*`)`
minimize a free resolution

`betty(`*resolution* | *list*`)`
compute the graded Betti numbers of a module represented by a resolution