CALI

A REDUCE Package for Commutative Algebra Version 2.2.1

Hans-Gert Gräbe

Universität Leipzig
Institut für Informatik
Augustusplatz 10 – 11
04109 Leipzig / Germany

email: graebe@informatik.uni-leipzig.de

June 28, 1995

Key words: affine and projective monomial curves, affine and projective sets of points, analytic spread, associated graded ring, blowup, border bases, constructive commutative algebra, dual bases, elimination, equidimensional part, extended Gröbner factorizer, free resolution, Gröbner algorithms for ideals and module, Gröbner factorizer, ideal and module operations, independent sets, intersections, lazy standard bases, local free resolutions, local standard bases, minimal generators, minors, normal forms, pfaffians, polynomial maps, primary decomposition, quotients, symbolic powers, symmetric algebra, triangular systems, weighted Hilbert series, primality test, radical, unmixed radical.

Contents

1 Introduction

This package contains algorithms for computations in commutative algebra closely related to the Gröbner algorithm for ideals and modules. Its heart is a new implementation of the Gröbner algorithm[1] that allows the computation of syzygies, too. This implementation is also applicable to submodules of free modules with generators represented as rows of a matrix.

Moreover CALI contains facilities for local computations, using a modern implementation of Mora's standard basis algorithm, see [26] and [13], that works for arbitrary term orders. The full analogy between modules over the local ring $k[x_v : v \in H]_{\mathbf{m}}$ and homogeneous (in fact H-local) modules over $k[x_v : v \in H]$ is reflected through the switch *Noetherian*. Turn it on (Gröbner basis, the default) or off (local standard basis) to choose appropriate algorithms automatically. In v. 2.2 we present an unified approach to both cases, using reduction with bounded ecart for non Noetherian term orders, see [14] for details. This allows to have a common driver for the Gröbner algorithm in both cases.

CALI extends also the restricted term order facilities of the `groebner` package, defining term orders by degree vector lists, and the rigid implementation of the sugar idea, by a more flexible *ecart* vector, in particular useful for local computations, see [13].

The package was designed mainly as a symbolic mode programming environment extending the build-in facilities of REDUCE for the computational approach to problems arising naturally in commutative algebra. An algebraic mode interface accesses (in a more rigid frame) all important features implemented symbolically and thus should be favored for short sample computations.

On the other hand, tedious computations are strongly recommended to be done symbolically since this allows considerably more flexibility and avoids unnecessary translations of intermediate results from CALI's internal data representation to the algebraic mode and vice versa. Moreover, one can easily extend the package with new symbolic mode scripts, or do more difficult interactive computations. For all these purposes the symbolic mode interface offers substantially more facilities than the algebraic one.

For a detailed description of special symbolic mode procedures one should consult the source code and the comments therein. In this manual we can give only a brief description of the main ideas incorporated into the package CALI. We concentrate on the data structure design and the description of the more advanced algorithms. For sample computations from several fields of commutative algebra the reader may consult also the *cali.tst* file.

As main topics CALI contains facilities for

- defining rings, ideals and modules,
- computing Gröbner bases and local standard bases,
- computing syzygies, resolutions and (graded) Betti numbers,

[1] The data representation even for polynomials is different from that given in the `groebner` package distributed with REDUCE (and rests on ideas used in the `dipoly` package).

- computing (now also weighted) Hilbert series, multiplicities, independent sets, and dimensions,
- computing normal forms and representations,
- computing sums, products, intersections, quotients, stable quotients, elimination ideals etc.,
- primality tests, computation of radicals, unmixed radicals, equidimensional parts, primary decompositions etc. of ideals and modules,
- advanced applications of Gröbner bases (blowup, associated graded ring, analytic spread, symmetric algebra, monomial curves etc.),
- applications of linear algebra techniques to zero dimensional ideals, as e.g. the FGLM change of term orders, border bases and affine and projective ideals of sets of points,
- splitting polynomial systems of equations mixing factorization and the Gröbner algorithm, triangular systems, and different versions of the extended Gröbner factorizer.

Below we will use freely without further explanation the notions common for text books and papers about constructive commutative algebra, assuming the reader to be familiar with the corresponding ideas and concepts. For further references see e.g. the text books [2], [7] and [21] or the survey papers [5], [6] and [27].

1.1 Description of the Documents Distributed with CALI

The CALI package contains the following files:

cali.chg

a detailed report of changes from v. 2.1 to v. 2.2. and 2.2.1

cali.log

the output file, that cali.tst should produce with

```
load_package cali;
out "logfile"$
in "cali.tst";
shut "logfile"$
```

cali.red

the CALI source file.

cali.tex

this manual.

cali.tst

a test file with various examples and applications of CALI.

CALI should be precompiled as usual, i.e. either using the *makefasl* utility of REDUCE or "by hand" via

```
faslout "cali"$
in "cali.red"$
faslend$
```

and then loaded via

```
load_package cali;
```

Upon successful loading CALI responds with a message containing the version number and the last update of the distribution.

> Feel free to contact me by email if You have problems to get CALI started. Also comments, hints, bug reports etc. are welcome.

1.2 CALI's Language Concept

From a certain point of view one of the major disadvantage of the current RLISP (and the underlying PSL) language is the fact that it supports modularity and data encapsulation only in a rudimentary way. Since all parts of code loaded into a session are visible all the time, name conflicts between different packages may occur, will occur (even not issuing a warning message), and are hard to prevent, since packages are developed (and are still developing) by different research groups at different places and different time.

A (yet rudimentary) concept of REDUCE packages and modules indicates the direction into what the REDUCE designers are looking for a solution for this general problem.

CALI (2.0 and higher) follows a name concept for internal procedures to mimick data encapsulation at a semantical level. We hope this way on the one hand to resolve the conflicts described above at least for the internal part of CALI and on the other hand to anticipate a desirable future and already foregoing development of REDUCE towards a true modularity.

The package CALI is divided into several modules, each of them introducing either a single new data type together with basic facilities, constructors, and selectors or a collection of algorithms subject to a common problem. Each module contains *internal procedures*, conceptually hidden by this module, *local procedures*, designed for a CALI wide use, and *global procedures*, exported by CALI into the general (algebraic or symbolic) environment of REDUCE. A header *module cali* contains all (fluid) global variables and switches defined by the pacakge CALI.

Along these lines the CALI procedures available in symbolic mode are divided into three types with the following naming convention:

`module!=procedure`

internal to the given module.

`module_procedure`

exported by the given module into the local CALI environment.

`procedure!*`

a global procedure usually having a semantically equivalent procedure (possibly with another parameter list) without trailing asterisk in algebraic mode.

There are also symbolic mode equivalents without trailing asterisk, if the algebraic procedure is not a *psopfn*, but a *symbolic operator*. They transfer data to CALI's internal structure and call the corresponding procedure with trailing asterisk. CALI 2.2 distinguishes between algebraic and symbolic calls of such a procedure. In symbolic mode such a procedure calls the corresponding procedure with trailing asterisk directly without data transfer.

CALI 2.2 follows also a more concise concept for global variables. There are three types of them:

True *fluid* global variables,

that are part of the current data structure, as e.g. the current base ring and the degree vector. They are often locally rebound to be restored after interrupts.

Global variables, stored on the property list of the package name `cali`,

that reflect the state of the computational model as e.g. the trace level, the output print level or the chosen version of the Gröbner basis algorithm. There are several such parameters in the module *dualbases* to serve the common dual basis driver with information for different applications.

Switches,

that allow to choose different branches of algorithms. Note that this concept interferes with the second one. Different *versions* of algorithms, that apply different functions in a common driver, are *not* implemented through switches.

1.3 New and Improved Facilities in v. 2.1

The major changes in v. 2.1 reflect the experience we've got from the use of CALI 2.0. The following changes are worth mentioning explicitely:

1. The algebraic rule concept was adapted to CALI. It allows to supply rule based coefficient domains. This is a more efficient way to deal with (easy) algebraic numbers than through the *arnum package*.

2. *listtest* and *listminimize* provide an unified concept for different list operations previously scattered in the source text.

3. There are several new quotient algorithms at the symbolic level (both the general element and the intersection approaches are available) and new features for the computation of equidimensional hull and equidimensional radical.

4. A new *module scripts* offers advanced applications of Gröbner bases.

5. Several advanced procedures initialize a Gröbner basis computation over a certain intermediate base ring or term order as e.g. *eliminate*, *resolve*, *matintersect* or all *primary decomposition* procedures. Interrupting a computation in v. 2.1 now restores the original values of CALI's global variables, since all intermediate procedures work with local copies of the global variables.[2] This doesn't apply to advanced procedures that change the current base ring as e.g. *blowup*, *preimage*, *sym* etc.

1.4 New and Improved Facilities in v. 2.2

Version 2.2 (beside bug fixes) incorporates several new facilities of constructive non linear algebra that we investigated the last two years, as e.g. dual bases, the Gröbner factorizer, triangular systems, and local standard bases. Essential changes concern the following topics:

1. The CALI modules *red* and *groeb* were rewritten and the *module mora* was removed. This is due to new theoretical insight into standard bases theory as e.g. described in [13] or [14]. The Gröbner basis algorithm is reorganized as a Gröbner driver with simplifier and base lists, that involves different versions of polynomial reduction according to the setting via *gbtestversion*. It applies now to both noetherian and non noetherian term orders in a unified way.

   The switches *binomial* and *lazy* were removed.

2. The Gröbner factorizer was thoroughly revised, extended along the lines explained in [15], and collected into a separate *module groebf*. It now allows a list of constraints also in algebraic mode. Two versions of an *extended Gröbner factorizer* produce *triangular systems*, i.e. a decomposition into quasi prime components, see [16], that are well suited for further (numerical) evaluation. There is also a version of the Gröbner factorizer that allows a list of problems as input. This is especially useful, if a system is splitted with respect to a "cheap" (e.g. degrevlex) term order and the pieces are recomputed with respect to a "hard" (e.g. pure lex) term order.

   The extended Gröbner factorizer involves, after change to dimension zero, the computation of *triangular systems*. The corresponding module *triang* extends the facilities for zero dimensional ideals and modules in the *module odim*.

3. A new *module lf* implements the *dual bases* approach as described in [20]. On this basis there are new implementations of *affine_points* and *proj_points*, that are significantly faster than the old ones. The linear algebra *change of term orders* [9] is

[2] Note that recovering the base ring this way may cause some trouble since the intermediate ring, installed with *setring*, changed possibly the internal variable order set by *setkorder*.

available, too. There are two versions, one with precomputed *border basis*, the other with conventional normal forms.

4. *dpmat*s now have a *gb-tag* that indicates, whether the given ideal or module basis is already a Gröbner basis. This avoids certain Gröbner basis recomputations especially during advanced algorithms as e.g. prime decomposition. In the algebraic interface Gröbner bases are computed automatically when needed rather than to issue an error message as in v. 2.1. So one can call *modequalp* or *dim* etc. not having computed Gröbner bases in advance. Note that such automatic computation can be avoided with *setgbasis*.

5. Hilbert series are now *weighted Hilbert series*, since e.g. for blow up rings the generating ideal is multigraded. Usual Hilbert series are computed as in v. 2.1 with respect to the *ecart vector*. Weighted Hilbert series accept a list of (integer) weight lists as second parameter.

6. There are some name and conceptual changes to existing procedures and variables to have a more concise semantic concept. This concerns

   tracing (the trace parameter is now stored on the property list of `cali` and should be set with *setcalitrace*),
   choosing different versions of the Gröbner algorithm (through *gbtestversion*) and the Hilbert series computation (through *hftestversion*),
   some names (*mat2list* replaced *flatten*, *HilbertSeries* replaced *hilbseries*) and
   parameter lists of some local and internal procedures (consult *cali.chg* for details).

7. The *revlex term order* is now the reverse lexicographic term order on the **reversely** ordered variables. This is consistent with other computer algebra systems (e.g. SINGULAR or AXIOM)[3] and implies the same order on the variables for deglex and degrevlex term orders (this was the main reason to change the definition).

8. Ideals of minors, pfaffians and related stuff are now implemented as extension of the internal `matrix` package and collected into a separate *module calimat*. Thus they allow more general expressions, especially with variable exponents, as general REDUCE matrices do. So one can define generic ideals as e.g. ideals of minors or pfaffians of matrices, containing generic expressions as elements. They must be specified for further use in CALI substituting general exponents by integers.

[3]But different to the currently distibuted `groebner` package in REDUCE. Note that the computations in [15] were done *before* these changes.

1.5 New and Improved Facilities in v. 2.2.1

The main change concerns the primary decomposition algorithm, where I fixed a serious bug for deciding, which embedded primes are really embedded[4]. During that remake I incorporated also the Gröbner factorizer to compute isolated primes. Since REDUCE has no multivariate *modular* factorizer, the switch *factorprimes* may be turned off to switch to the former algorithm.

Some minor bugs are fixed, too, e.g. the bug that made *radical* crashing.

2 The Computational Model

This section gives a short introduction into the data type design of CALI at different levels. First (§1 and 2) we describe CALI's way of algorithmic translation of the abstract algebraic objects *ring of polynomials, ideal* and (finitely generated) *module*. Then (§3 and 4) we describe the algebraic mode interface of CALI and the switches and global variables to drive a session. In the next chapter we give a more detailed overview of the basic (symbolic mode) data structures involved with CALI. We refer to the appendix for a short summary of the commands available in algebraic mode.

2.1 The Base Ring

A polynomial ring consists in CALI of the following data:

a list of variable names

: All variables not occuring in the list of ring names are treated as parameters. Computations are executed denominatorfree, but the results are valid only over the corresponding parameter *field* extension.

a term order and a term order tag

: They describe the way in which the terms in each polynomial (and polynomial vector) are ordered.

an ecart vector

: A list of positive integers corresponding to the variable names.

A *base ring* may be defined (in algebraic mode) through the command

```
setring <ring>
```

with $< ring > ::= \{$ vars, tord, tag [, ecart] $\}$ resp.

```
setring(vars, tord, tag [,ecart])
```

[4]That there must be a bug was pointed out to me by Shimoyama Takeshi who compared different p.d. implementations. The bug is due to an incorrect test for embedded primes: A (superfluous) primary component may contain none of the isolated primary components, but their intersection! Note that neither [10] nor [2] comment on that. Details of the implementation will appear in [17].

This sets the global (symbolic) variable *cali!=basering*. Here `vars` is the list of variable names, `tord` a (possibly empty) list of weight lists, the *degree vectors*, and `tag` the tag LEX or REVLEX. Optionally one can supply `ecart`, a list of positive integers of the same length as `vars`, to set an ecart vector different from the default one (see below).

The degree vectors must have the same length as `vars`. If $(w_1 \ \ldots \ w_k)$ is the list of degree vectors then

$$\begin{array}{lllll} x^a < x^b & :\Leftrightarrow & \text{either} & w_j(x^a) = w_j(x^b) \quad \text{for } j < i & \text{and} \\ & & & w_i(x^a) < w_i(x^b) & \\ & & \text{or} & w_j(x^a) = w_j(x^b) \quad \text{for all } j & \text{and} \\ & & & x^a <_{lex} x^b \text{ resp. } x^a <_{revlex} x^b & \end{array}$$

Here $<_{lex}$ resp. $<_{revlex}$ denote the *lexicographic* (tag=LEX) resp. *reverse lexicographic* (tag=REVLEX) term orders[5] with respect to the variable order given in `vars`, i.e.

$$x^a < x^b \quad :\Leftrightarrow \quad \exists\, j\ \forall\, i < j\ :\ a_i = b_i \quad \text{and} \quad a_j < b_j \text{ (lex.)}$$

or

$$x^a < x^b \quad :\Leftrightarrow \quad \exists\, j\ \forall\, i > j\ :\ a_i = b_i \quad \text{and} \quad a_j > b_j \text{ (revlex.)}$$

Every term order can be represented in such a way, see [24].

During the ring setting the term order will be checked to be Noetherian (i.e. to fulfill the descending chain condition) provided the *switch Noetherian* is on (the default). The same applies turning *noetherian on*: If the term order of the underlying base ring isn't Noetherian the switch can't be turned over. Hence, starting from a non Noetherian term order, one should define *first* a new ring and *then* turn the switch on.

Useful term orders can be defined by the procedures

`degreeorder vars`,

that returns $tord = \{\{1, \ldots, 1\}\}$.

`localorder vars`,

that returns $tord = \{\{-1, \ldots, -1\}\}$ (a non Noetherian term order for computations in local rings).

`eliminationorder(vars,elimvars)`,

that returns a term order for elimination of the variables in `elimvars`, a subset of all `vars`. It's recommended to combine it with the tag REVLEX.

`blockorder(vars,integerlist)`,

that returns the list of degree vectors for the block order with block lengths given in the `integerlist`. Note that these numbers should sum up to the length of the variable list supplied as the first argument.

[5]The definition of the revlex term order changed for version 2.2.

Examples:

```
vars:={x,y,z};
tord:=degreeorder vars;   % Returns {{1,1,1}}.
setring(vars,tord,lex);   % GRADLEX in the groebner package.

% or

setring({a,b,c,d},{},lex); % LEX in the groebner package.

% or

vars:={a,b,c,x,y,z};
tord:=eliminationorder(vars,{x,y,z});
tord:=reverse blockorder(vars,{3,3});
                          % Return both {{0,0,0,1,1,1},{1,1,1,0,0,0}}.
setring(vars,tord,revlex);
```

The base ring is initialized with

```
{{t,x,y,z},{{1,1,1,1}},revlex,{1,1,1,1}},
```

i.e. $S = k[t, x, y, z]$ supplied with the degree wise reverse lexicographic term order.

`getring m`

returns the ring attached to the object with the identifier m. E.g.

`setring getring m`

(re)sets the base ring to the base ring of the formerly defined object (ideal or module) m.

`getring()`

returns the currently active base ring.

CALI defines also an *ecart vector*, attaching to each variable a positive weight with respect to that homogenizations and related algorithms are executed. It may be set optionally by the user during the *setring* command. (Default: If the term order is a (positive) degree order then the ecart is the first degree vector, otherwise each ecart equals 1).

The ecart vector is used in several places for efficiency reason (Gröbner basis computation with the sugar strategy) or for termination (local standard bases). If the input is homogeneous the ecart vector should reflect this homogeneity rather than the first degree vector to obtain the best possible performance. For a discussion of local computations with encoupled ecart vector see [13]. In general the ecart vector is recommended to be chosen in such a way that the input examples become close to be homogeneous. *Homogenizations* and *Hilbert series* are computed with respect to this ecart vector.

`getecart()` returns the ecart vector currently set.

2.2 Ideals and Modules

If $S = k[x_v,\ v \in H]$ is a polynomial ring, a matrix M of size $r \times c$ defines a map

$$f\ :\ S^r \longrightarrow S^c$$

by the following rule

$$f(v) := v \cdot M \qquad \text{for } v \in S^r.$$

There are two modules, connected with such a map, $im\ f$, the submodule of S^c generated by the rows of M, and $coker\ f$ $(= S^c/im\ f)$. Conceptually we will identify M with $im\ f$ for the basic algebra, and with $coker\ f$ for more advanced topics of commutative algebra (Hilbert series, dimension, resolution etc.) following widely accepted conventions.

With respect to a fixed basis $\{e_1, \ldots, e_c\}$ one can define module term orders on S^c, Gröbner bases of submodules of S^c etc. They generalize the corresponding notions for ideal bases. See [8] or [22] for a detailed introduction to this area of computational commutative algebra. This allows to define joint facilities for both ideals and submodules of free modules. Moreover computing syzygies the latter come in in a natural way.

CALI handles ideal and module bases in a unique way representing them as rows of a *dpmat* (**d**istributive **p**olynomial **mat**rix). It attaches to each unit vector e_i a monomial x^{a_i}, the i-th *column degree* and represents the rows of a dpmat M as lists of module terms $x^a e_i$, sorted with respect to a *module term order*, that may be roughly[6] described as

$$x^a e_i < x^b e_j \quad :\Leftrightarrow \quad \text{either} \qquad x^a x^{a_i} < x^b x^{a_j} \text{ in } S$$

$$\text{or} \qquad x^a x^{a_i} = x^b x^{a_j}$$

$$\text{and}$$

$$i < j \text{ (lex.) resp. } i > j \text{ (revlex.)}$$

Every dpmat M has its own column degrees (no default !). They are managed through a global (symbolic) variable *cali!=degrees*.

`getdegrees m`

returns the column degrees of the object with identifier m.

`getdegrees()`

returns the current setting of *cali!=degrees*.

`setdegrees <list of monomials>`

sets *cali!=degrees* correspondingly. Use this command before executing *setmodule* to give a dpmat prescribed column degrees since cali!=degrees has no default value and changes during computations. A good guess is to supply the empty list (i.e. all column degrees are equal to $\mathbf{x}^0$). Be careful defining modules without prescribed column degrees.

[6] The correct definition is even more difficult.

To distinguish between *ideals* and *modules* the former are represented as a *dpmat* with $c = 0$ (and hence without column degrees). If $I \subset S$ is such an ideal one has to distinguish between the ideal I (with $c = 0$, allowing special ideal operations as e.g. ideal multiplication) and the submodule I of the free one dimensional module S^1 (with $c = 1$, allowing matrix operations as e.g. transposition, matrix multiplication etc.). *ideal2mat* converts an (algebraic) list of polynomials into an (algebraic) matrix column whereas *mat2list* collects all matrix entries into a list.

2.3 The Algebraic Mode Interface

Corresponding to CALI's general philosophy explained in the introduction the algebraic mode interface translates algebraic input into CALI's internal data representation, calls the corresponding symbolic functions, and retranslates the result back into algebraic mode. Since Gröbner basis computations may be very tedious even on small examples, one should find a well balance between the storage of results computed earlier and the unavoidable time overhead and memory request associated with the management of these results.

Therefore CALI distinguishes between *free* and *bounded* identifiers. Free identifiers stand only for their value whereas to bounded identifiers several internal information is attached to their property list for later use.

After the initialization of the *base ring* bounded identifiers for ideals or modules should be declared via

```
setmodule(name,matrix value)
```

resp.

```
setideal(name,list of polynomials)
```

This way the corresponding internal representation (as *dpmat*) is attached to `name` as the property *basis*, the prefix form as its value and the current base ring as the property *ring*.

Performing any algebraic operation on objects defined this way their ring will be compared with the current base ring (including the term order). If they are different an error message occurs. If `m` is a valid name, after resetting the base ring

```
setmodule(m1,m)
```

reevaluates `m` with respect to the new base ring (since the *value* of `m` is its prefix form) and assigns the reordered dpmat to `m1` clearing all information previously computed for `m1` (`m1` and `m` may coincide).

All computations are performed with respect to the ring $S = k[x_v \in \texttt{vars}]$ over the field k. Nevertheless by efficiency reasons *base coefficients* are represented in a denominator free way as standard forms. Hence the computational properties of the base coefficient domain depend on the *dmode* and also on auxiliary variables, contained in the expressions, but not in the variable list. They are assumed to be parameters.

Best performance will be obtained with integer or modular domain modes, but one can also try *algebraic numbers* as coefficients as e.g. generated by `sqrt` or the `arnum` package.

To avoid an unnecessary slow-down connected with the management of simplified algebraic expressions there is a *switch hardzerotest* (default: off) that may be turned on to force an additional simplification of algebraic coefficients during each zero test. It should be turned on only for domain modes without canonical representations as e.g. mixtures of arnums and square roots. We remind the general zero decision problem for such domains.

Alternatively, CALI offers the possibility to define a set of algebraic substitution rules that will affect CALI's base coefficient arithmetic only.

`setrules <rule list>`

transfers the (algebraic) rule list into the internal representation stored at the `cali` value `rules`.
In particular, `setrules {}` clears the rules previously set.

`getrules()`

returns the internal CALI rules list in algebraic form.

We recommend to use *setrules* for computations with algebraic numbers since they are better adapted to the data structure of CALI than the algebraic numbers provided by the `arnum` package. Note, that due to the zero decision problem complicated *setrules* based computations may produce wrong results if base coefficient's pseudo division is involved (as e.g. with *dp_pseudodivmod*). In this case we recommend to enlarge the variable set and add the defining equations of the algebraic numbers to the equations of the problem[7].

The standard domain (Integer) doesn't allow denominators for input. *setideal* clears automatically the common denominator of each input expression whereas a polynomial matrix with true rational coefficients will be rejected by *setmodule*.

One can save/initialize ideal and module bases together with their accompanying data (base ring, degrees) to/from a file:

```
savemat(m,name)
```

resp.

```
initmat name
```

execute the file transfer from/to disk files with the specified file `name`. e.g.

```
savemat(m,"myfile");
```

saves the base ring and the ideal basis of m to the file "myfile" whereas

```
setideal(m,initmat "myfile");
```

sets the current base ring (via a call to *setring*) to the base ring of m saved at "myfile" and then recovers the basis of m from the same file.

[7] A *qring* facility for the computation over quotient rings will be incorporated into future versions.

2.4 Switches and Global Variables

There are several switches, (fluid) global variables, a trace facility, and global parameters on the property list of the package name `cali` to control CALI's computations.

Switches

bcsimp

on: Cancel out gcd's of base coefficients. (Default: on)

detectunits

on: replace polynomials of the form
$\langle monomial \rangle * \langle polynomial\ unit \rangle$ by $\langle monomial \rangle$ during interreductions and standard basis computations.
Affects only local computations. (Default: off)

factorprimes

on: Invoke the Gröbner factorizer during computation of isolated primes. (Default: on). Note that REDUCE lacks a modular multivariate factorizer, hence for modular prime decomposition computations this switch has to be turned off.

factorunits

on: factor polynomials and remove polynomial unit factors during interreductions and standard basis computations.
Affects only local computations. (Default: off)

hardzerotest

on: try an additional algebraic simplification of base coefficients at each base coefficient's zero test. Useful only for advanced base coefficient domains without canonical REDUCE representation. May slow down the computation drastically. (Default: off)

lexefgb

on: Use the pure lexicographic term order and *zerosolve* during reduction to dimension zero in the *extended Gröbner factorizer*. This is a single, but possibly hard task compared to the degrevlex invocation of *zerosolve1*. See [16] for a discussion of different zero dimensional solver strategies. (Default: off)

Noetherian

on: choose algorithms for Noetherian term orders.
off: choose algorithms for local term orders.
(Default: on)

red_total

on: compute total normal forms, i.e. apply reduction (Noetherian term orders) or reduction with bounded ecart (non Noetherian term orders to tail terms of polynomials, too.
off: Do only top reduction.
(Default: on)

Tracing

Different to v. 2.1 now intermediate output during the computations is controlled by the value of the `trace` and `printterms` entries on the property list of the package name `cali`. The former value controls the intensity of the intermediate output (Default: 0, no tracing), the latter the number of terms printed in such intermediate polynomials (Default: all).

`setcalitrace <n>`

changes the trace intensity. Set $n = 2$ for a sparse tracing (a dot for each reduction step). Other good suggestions are the values 30 or 40 for tracing the Gröbner algorithm or $n > 70$ for tracing the normal form algorithm. The higher n the more intermediate information will be given.

`setcaliprintterms <n>`

sets the number of terms that are printed in intermediate polynomials. Note that this does not affect the output of whole *dpmats*. The output of polynomials with more than n terms ($n > 0$) breaks off and continues with ellipses.

`clearcaliprintterms()`

clears the `printterms` value forcing full intermediate output (according to the current trace level).

Global Variables

cali!=basering

The currently active base ring initialized e.g. by *setring*.

cali!=degrees

The currently active module component degrees initialized e.g. by *setdegrees*.

cali!=monset

A list of variable names considered as non zero divisors during Gröbner basis computations initialized e.g. by *setmonset*. Useful e.g. for binomial ideals defining monomial varieties or other prime ideals.

Entries on the Property List of `cali`

This approach is new for v. 2.2. Information concerning the state of the computational model as e.g. trace intensity, base coefficient rules, or algorithm versions are stored as values on the property list of the package name *cali*. This concerns

trace and *printterms*

see above.

efgb

Changed by the *switch lexefgb*.

groeb!=rf

Reduction function invoked during the Gröbner algorithm. It can be changed with *gbtestversion* $< n >$ ($n = 1, 2, 3$, default is 1).

hf!=hf

Variant for the computation of the Hilbert series numerator. It can be changed with *hftestversion* $< n >$ ($n = 1, 2$, default is 1).

rules

Algebraic "replaceby" rules introduced to CALI with the *setrules* command.

evlf, *varlessp*, *sublist*, *varnames*, *oldborderbasis*, *oldring*, *oldbasis*

see *module lf*, implementing the dual bases approach.

3 Basic Data Structures

In the following we describe the data structure layers underlying the dpmat representation in CALI and some important (symbolic) procedures to handle them. We refer to the source code and the comments therein for a more complete survey about the procedures available for different data types.

3.1 The Coefficient Domain

Base coefficients as implemented in the *module bcsf* are standard forms in the variables outside the variable list of the current ring. All computations are executed "denominator free" over the corresponding quotient field, i.e. gcd's are canceled out without request. To avoid this set the *switch bcsimp* off.[8] In the given implementation we use the s.f. procedure *qremf* for effective divisibility test. We had some trouble with it under *on factor*.

Additionally it is possible to supply the parameters occuring as base coefficients with a (global) set of algebraic rules.[9]

[8] This induces a rapid base coefficient's growth and doesn't yield **Z**-Gröbner bases in the sense of [10] since the S-pair criteria are different.

[9] This is different from the LET rule mechanism since they must be present in symbolic mode. Hence for a simultaneous application of the same rules in algebraic mode outside CALI they must additionally be declared in the usual way.

`setrules!* r`

converts an algebraic mode rules list r as e.g. used in WHERE statements into the internal CALI format.

3.2 The Base Ring

The *base ring* is defined by its `name list`, the `degree matrix` (a list of lists of integers), the `ring tag` (LEX or REVLEX), and the `ecart`. The name list contains a phantom name `cali!=mk` for the module component at place 0.

The *module ring* exports among others the selectors *ring_names*, *ring_degrees*, *ring_tag*, *ring_ecart*, the test function *ring_isnoetherian* and the transfer procedures from/to an (appropriate, printable by *mathprint*) algebraic prefix form *ring_from_a* (including extensive tests of the supplied parameters for consistency) and *ring_2a*.

The following procedures allow to define a base ring:

`ring_define(name list, degree matrix, ring tag, ecart)`

combines the given parameters to a ring.

`setring!* <ring>`

sets *cali!=basering* and checks for consistency with the *switch Noetherian*. It also sets through *setkorder* the current variable list as main variables. It is strongly recommended to use *setring!* ...* instead of *cali!=basering:=...*.

`degreeorder!*` , `localorder!*`, `eliminationorder!*`, and `blockorder!*` define term order matrices in full analogy to algebraic mode.

There are three ring constructors for special purposes:

`ring_sum(a,b)`

returns a ring, that is constructed in the following way: Its variable list is the union of the (disjoint) lists of the variables of the rings a and b (in this order) whereas the degree list is the union of the (appropriately shifted) degree lists of b and a (in this order). The ring tag is that of a. Hence it returns (essentially) the ring $b \bigoplus a$ if b has a degree part (e.g. useful for elimination problems, introducing "big" new variables) and the ring $a \bigoplus b$ if b has no degree part (introducing "small" new variables).

`ring_rlp(r,u)`

u is a subset of the names of the ring r. Returns the ring r, but with a term order "first degrevlex on u, then the order on r".

`ring_lp(r,u)`

As *rlp*, but with a term order "first lex on u, then the order on r".

Example:

```
vars:='(x y z)
setring!* ring_define(vars,degreeorder!* vars,'lex,'(1 1 1));
                        % GRADLEX in the groebner package.
```

3.3 Monomials

The current version uses a place-driven exponent representation closely related to a vector model. This model handles term orders on S and module term orders on S^c in a unique way. The zero component of the exponent list of a monomial contains its module component (> 0) or 0 (ring element). All computations are executed with respect to a *current ring* (*cali!=basering*) and *current (monomial) weights* of the free generators $e_i, i = 1, \ldots, c$, of S^c (*cali!=degrees*). For efficiency reasons every monomial has a precomputed degree part that should be reevaluated if `cali!=basering` (i.e. the term order) or `cali!=degrees` were changed. `cali!=degrees` contains the list of column degrees of the current module as an assoc. list and will be set automatically by (almost) all dpmat procedure calls. Since monomial operations use the degree list that was precomputed with respect to fixed column degrees (and base ring)

> **watch carefully for `cali!=degrees` programming at the monomial or dpoly level !**

As procedures there are selectors for the module component, the exponent and the degree parts, comparison procedures, procedures for the management of the module component and the degree vector, monomial arithmetic, transfer from/to prefix form, and more special tools.

3.4 Polynomials and Polynomial Vectors

CALI uses a distributive representation as a list of terms for both polynomials and polynomial vectors, where a *term* is a dotted pair

$$(< monomial > \ . \ < base\ coefficient >).$$

The *ecart* of a polynomial (vector) $f = \sum t_i$ with (module) terms t_i is defined as

$$max(ec(t_i)) - ec(lt(t_i)),$$

see [13]. Here $ec(t_i)$ denotes the ecart of the term t_i, i.e. the scalar product of the exponent vector of t_i (including the monomial weight of the module generator) with the ecart vector of the current base ring.

As procedures there are selectors, dpoly arithmetic including the management of the module component, procedures for reordering (and reevaluating) polynomials wrt. new term order degrees, for extracting common base coefficient or monomial factors, for transfer from/to prefix form and for homogenization and dehomogenization (wrt. the current ecart vector).

Two advanced procedures use ideal theory ingredients:

`dp_pseudodivmod(g,f)`

returns a dpoly list $\{q, r, z\}$ such that $z \cdot g = q \cdot f + r$ and z is a dpoly unit (i.e. a scalar for Noetherian term orders). For non Noetherian term orders the necessary modifications are described in [14].
g, f and r belong to the same free module or ideal.

`dpgcd(a,b)`

computes the gcd of two dpolys a and b by the syzygy method: The syzygy module of $\{a, b\}$ is generated by a single element $[-b_0 \;\; a_0]$ with $a = ga_0, b = gb_0$, where g is the gcd of a and b. Since it uses dpoly pseudodivision it may work not properly with *setrules*.

3.5 Base Lists

Ideal bases are one of the main ingredients for dpmats. They are represented as lists of *base elements* and contain together with each dpoly entry the following information:

- a number (the row number of the polynomial vector in the corresponding dpmat).
- the dpoly, its ecart (as the main sort criterion), and length.
- a representation part, that may contain a representation of the given dpoly in terms of a certain fixed basis (default: empty).

The representation part is managed during normal form computations and other row arithmetic of dpmats appropriately with the following procedures:

`bas_setrelations b`

sets the relation part of the base element i in the base list b to e_i.

`bas_removerelations b`

removes all relations, i.e. replaces them with the zero polynomial vector.

`bas_getrelations b`

gets the relation part of b as a separate base list.

Further there are procedures for selection and construction of base elements and for the manipulation of lists of base elements as e.g. sorting, renumbering, reordering, simplification, deleting zero base elements, transfer from/to prefix form, homogenization and dehomogenization.

3.6 Dpoly Matrices

Ideals and matrices, represented as *dpmat*s, are the central data type of the CALI package, as already explained above. Every dpmat m combines the following information:

- its size (*dpmat_rows* m,*dpmat_cols* m),

- its base list (*dpmat_list* m) and
- its column degrees as an assoc. list of monomials (*dpmat_coldegs* m). If this list is empty, all degrees are assumed to be equal to x^0.
- New in v. 2.2 there is a *gb-tag* (*dpmat_gbtag* m), indicating that the given base list is already a Gröbner basis (under the given term order).

The *module dpmat* contains selectors, constructors, and the algorithms for the basic management of this data structure as e.g. file transfer, transfer from/to algebraic prefix forms, reordering, simplification, extracting row degrees and leading terms, dpmat matrix arithmetic, homogenization and dehomogenization.

The modules *matop* and *quot* collect more advanced procedures for the algebraic management of dpmats.

3.7 Extending the REDUCE Matrix Package

In v. 2.2 minors, Jacobian matrix, and Pfaffians are available for general REDUCE matrices. They are collected in the *module calimat* and allow to define procedures in more generality, especially allowing variable exponents in polynomial expressions. Such a generalization is especially useful for the investigation of whole classes of examples that may be obtained from a generic one by specialization. In the following m is a matrix given in algebraic prefix form.

`matjac(m,l)`

returns the Jacobian matrix of the ideal m (given as an algebraic mode list) with respect to the variable list l.

`minors(m,k)`

returns the matrix of k-minors of the matrix m.

`ideal_of_minors(m,k)`

returns the ideal of the k-minors of the matrix m.

`pfaffian m`

returns the pfaffian of a skewsymmetric matrix m.

`ideal_of_pfaffians(m,k)`

returns the ideal of the $2k$-pfaffians of the skewsymmetric matrix m.

`random_linear_form(vars,bound)`

returns a random linear form in algebraic prefix form in the supplied variables *vars* with integer coefficients bounded by the supplied *bound*.

`singular_locus!*(m,c)`

returns the singular locus of m (as dpmat). m must be an ideal of codimension c given as a list of polynomials in prefix form. `Singular_locus` computes the ideal generated by the corresponding Jacobian and m itself.

4 About the Algorithms Implemented in CALI

Below we give a short explanation of the main algorithmic ideas of CALI and the way they are implemented and may be accessed (symbolically).

4.1 Normal Form Algorithms

For v. 2.2 we completely revised the implementation of normal form algorithms due to the insight obtained from our investigations of normal form procedures for local term orders in [14] and [13]. It allows a common handling of Noetherian and non Noetherian term orders already on this level thus making superfluous the former duplication of reduction procedures in the modules *red* and *mora* as in v. 2.1.

Normal form algorithms reduce polynomials (or polynomial vectors) with respect to a given finite set of generators of an ideal or module. The result is not unique except for a total normal form with respect to a Gröbner basis. Furthermore different reduction strategies may yield significant differences in computing time.

CALI reduces by first matching, usually keeping base lists sorted with respect to the sort predicate *red_better*. In v. 2.2 we sort solely by the dpoly length, since the introduction of *red_TopRedBE*, i.e. reduction with bounded ecart, guarantees termination also for non Noetherian term orders. Overload red_better for other reduction strategies.

Reduction procedures produce for a given ideal basis $B \subset S$ and a polynomial $f \in S$ a (pseudo) normal form $h \in S$ such that $h \equiv u \cdot f \ mod \ B$ where $u \in S$ is a polynomial unit, i.e. a (polynomially represented) non zero domain element in the Noetherian case (pseudodivision of f by B) or a polynomial with a scalar as leading term in the non Noetherian case. Following up the reduction steps one can even produce a presentation of $h - u \cdot f$ as a polynomial combination of the base elements in B.

More general, given for $f_i \in B$ and f representations $f_i = \sum r_{ik} e_k = R_i \cdot E^T$ and $f = R \cdot E^T$ as polynomial combinations wrt. a fixed basis E one can produce such a presentation also for h. For this purpose the dpoly f and its representation are collected into a base element and reduced simultaneously by the base list B, that collects the base elements and their representations.

The main procedures of the newly designed reduction package are the following:

`red_TopRedBE(bas,model)`

> Top reduction with bounded ecart of the base element *model* by the base list *bas*, i.e. only reducing the top term and only with base elements with ecart bounded by that of *model*.

`red_TopRed(bas,model)`

> Top reduction of *model*, but without restrictions.

`red_TailRed(bas,model)`

> Make tail reduction on *model*, i.e. top reduction on the tail terms. For convergence this uses reduction with bounded ecart for non Noetherian term orders and full reduction otherwise.

There is a common *red_TailRedDriver* that takes a top reduction function as parameter. It can be used for experiments with other top reduction procedure combinations.

`red_TotalRed(bas,model)`

A terminating total reduction, i.e. for Noetherian term orders the classical one and for local term orders using tail reduction with bounded ecart.

`red_Straight bas`

Reduce (with *red_TailRed*) the tails of the polynomials in the base list *bas*.

`red_TopInterreduce bas`

Reduces the base list *bas* with *red_TopRed* until it has pairwise incomparable leading terms, computes correct representation parts, but does no tail reduction.

`red_Interreduce bas`

Does top and, if `on red_total`, also tail interreduction on the base list *bas*.

Usually, e.g. for ideal generation problems, there is no need to care about the multiplier u. If nevertheless one needs its value, the base element f may be prepared with *red_prepare* to collect this information in the 0-slot of its representation part. Extract this information with *red_extract*.

`red_redpol(bas,model)`

combines this tool with a total reduction of the base element *model* and returns a dotted pair

$$(< \textit{reduced model} > . < \textit{dpoly unit multiplier} >).$$

Advanced applications call the interfacing procedures

`interreduce!* m`

that returns an interreduced basis of the dpmat m.

`mod!*(f,m)`

that returns the dotted pair $(h.u)$ where h is the pseudo normal form of the dpoly f modulo the dpmat m and u the corresponding polynomial unit multiplier.

`normalform!*(a,b)`

that returns $\{a_1, r, z\}$ with $a_1 = z * a - r * b$ where the rows of the dpmat a_1 are the normalforms of the rows of the dpmat a with respect to the dpmat b.

For local standard bases the ideal generated by the basic polynomials may have components not passing through the origin. Although they do not contribute to the ideal in $Loc(S) = S_{\mathbf{m}}$ they usually heavily increase the necessary computational effort. Hence for local term orders one should try to remove polynomial units as soon as they are detected. To remove them from base elements in an early stage of the computation one can either try the (cheap) test, whether $f \in S$ is of the form $\langle monomial \rangle * \langle polynomial\ unit \rangle$ or factor f completely and remove polynomial unit factors. For base elements this may be done with *bas_detectunits* or *bas_factorunits*.

Moreover there are two switches *detectunits* and *factorunits*, both off by default, that force such automatic simplifications during more advanced computations.

The procedure *deleteunits!** tries explicitely to factor the basis polynomials of a dpmat and to remove polynomial units occuring as one of the factors.

4.2 The Gröbner and Standard Basis Algorithms

There is now a unique *module groeb* that contains the Gröbner resp. standard basis algorithms with syzygy computation facility and related topics. There are common procedures (working for both Noetherian and non Noetherian term orders)

`gbasis!* m`

that returns a minimal Gröbner or standard basis of the dpmat m,

`syzygies!* m`

that returns an interreduced basis of the first syzygy module of the dpmat m and

`syzygies1!* m`

that returns a (not yet interreduced) basis of the syzygy module of the dpmat m.

These procedures start the outer Gröbner engine (now also common for both Noetherian and non Noetherian term orders)

`groeb_stbasis(m,mgb,ch,syz)`

that returns, applied to the dpmat m, three dpmats g, c, s with

g — the minimal reduced Gröbner basis of m if $mgb = t$,

c — the transition matrix $g = c \cdot m$ if $ch = t$, and

s — the (not yet interreduced) syzygy matrix of m if $syz = t$.

The next layer manages the preparation of the representation parts of the base elements to carry the syzygy information, calls the *general internal driver*, and extracts the relevant information from the result of that computation. The general internal driver branches according to different reduction functions into several versions. Upto now there are three different strategies for the reduction procedures for the S-polynomial reduction (different versions may be chosen via *gbtestversion*):

1. Total reduction with local simplifier lists. For local term orders this is (almost) Mora's first version for the tangent cone (the default).

2. Total reduction with global simplifier list. For local term orders this is (almost) Mora's SimpStBasis, see [26].

3. Total reduction with bounded ecart.

The first two versions (almost) coincide for Noetherian term orders. The third version reduces only with bounded ecart, thus forcing more pairs to be treated than necessary, but usually less expensive to be reduced. It is not yet well understood, whether this idea is of practical importance.

groeb_lazystbasis calls the lazy standard basis driver instead, that implements Mora's lazy algorithm, see [26]. As *groeb_homstbasis*, the computation of Gröbner and standard bases via homogenization (Lazard's approach), it is not fully integrated into the algebraic interface. Use

`homstbasis!* m`

for the invocation of the homogenization approach to compute a standard basis of the dpmat m and

`lazystbasis!* m`

for the lazy algorithm.

Experts commonly agree that the classical approach is better for "computable" examples, but computations done by the author on large examples indicate, that both approaches are in fact independent.

The pair list management uses the sugar strategy, see [11], with respect to the current ecart vector. If the input is homogeneous and the ecart vector reflects this homogeneity then pairs are sorted by ascending degree. Hence no superfluous base elements will be computed in this case. In general the sugar strategy performs best if the ecart vector is chosen to make the input close to be homogeneous.

There is another global variable *cali!=monset* that may contain a list of variable names (a subset of the variable names of the current base ring). During the "pure" Gröbner algorithm (without syzygy and representation computations) common monomial factors containing only these variables will be canceled out. This shortcut is useful if some of the variables are known to be non zero divisors as e.g. in most implicitation problems.

`setmonset!* vars`

initializes *cali!=monset* with a given list of variables *vars*.

The Gröbner tools as e.g. pair criteria, pair list update, pair management and S-polynomial construction are available.

`groeb_mingb m`

extracts a minimal Gröbner basis from the dpmat m, removing base elements with leading terms, divisible by other leading terms.

`groeb_minimize(bas,syz)`

minimizes the dpmat pair (bas, syz) deleting superfluous base elements from *bas* using syzygies from *syz* containing unit entries.

4.3 The Gröbner Factorizer

If $\bar{k}$ is the algebraic closure of k, $B := \{f_1, \ldots, f_m\} \subset S$ a finite system of polynomials, and $C := \{g_1, \ldots, g_k\}$ a set of side conditions define the *relative set of zeroes*

$$Z(B, C) := \{a \in \bar{k}^n : \forall\ f \in B\ f(a) = 0 \text{ and } \forall g \in C\ g(a) \neq 0\}.$$

Its Zariski closure is the zero set of $I(B) :< \prod C >$.

The Gröbner factorizer solves the following problem:

Find a collection (B_α, C_α) of Gröbner bases B_α and side conditions C_α such that

$$Z(B, C) = \bigcup_\alpha Z(B_\alpha, C_\alpha).$$

The *module groebf* and the *module triang* contain algorithms related to that problem, triangular systems, and their generalizations as described in [15] and [16]. V. 2.2 thus heavily extends the algorithmic possibilities that were implemented in former releases of CALI.

Note that, different to v. 2.1, we work with constraint *lists*.

`groebfactor!*(bas,con)`

returns for the dpmat ideal *bas* and the constraint list *con* (of dpolys) a minimal list of $(dpmat, constraint\ list)$ pairs with the desired property.

During a preprocessing it splits the submitted basis *bas* by a recursive factorization of polynomials and interreduction of bases into a (reduced) list of smaller subproblems consisting of a partly computed Gröbner basis, a constraint list, and a list of pairs not yet processed. The main procedure forces the next subproblem to be processed until another factorization is possible. Then the subproblem splits into subsubproblems, and the subproblem list will be updated. Subproblems are kept sorted with respect to their expected dimension *easydim* forcing this way a *depth first* recursion. Returned and not yet interreduced Gröbner bases are, after interreduction, subject to another call of the preprocessor since interreduced polynomials may factor anew.

`listgroebfactor!* l`

proceeds a whole list of dpmats (without constraints) at once and strips off constraints at the end.

Using the (ordinary) Gröbner factorizer even components of different dimension may keep gluing together. The *extended Gröbner factorizer* involves a postprocessing, that guarantees a decomposition into puredimensional components, given by triangular systems instead of Gröbner bases. Triangular systems in positive dimension must not be Gröbner bases of the underlying ideal. They should be preferred, since they are more simple but contain all information about the (quasi) prime component that they represent. The complete Gröbner basis of the corresponding component can be obtained by an easy stable quotient computation, see [16]. We refer to the same paper for the definition of *triangular systems* in positive dimension, that is consistent with our approach.

`extendedgroebfactor!*(bas,c)` and `extendedgroebfactor1!*(bas,c)`

return a list of results $\{b_i, c_i, v_i\}$ in algebraic prefix form such that b_i is a triangular set wrt. the variables v_i and c_i is a list of constraints, such that $b_i :< \prod c_i >$ is the (puredimensional) recontraction of the zerodimensional ideal $b_i \bigotimes_k k(v_i)$. For the first version the recontraction is not computed, hence the output may be not minimal. The second version computes recontractions to decide superfluous components already during the algorithm. Note that the stable quotient computation involved for that purpose may drastically slow down the whole attempt.

The postprocessing involves a change to dimension zero and invokes (zero dimensional) triangular system computations from the *module triang*. In a first step *groebf_zeroprimes1* incorporates the square free parts of certain univariate polynomials into these systems and strips off the constraints (since relative sets of zeroes in dimension zero are Zariski closed), using a splitting approach analogous to the Gröbner factorizer. In a second step, according to the *switch lexefgb*, either *zerosolve!** or *zerosolve1!** converts these intermediate results into lists of triangular systems in prefix form. If *lexefgb* is `off` (the default), the zero dimensional term order is degrevlex and *zerosolve1!**, the "slow turn to lex" is involved, for `on lexefgb` the pure lexicographic term order and *zerosolve!**, Möllers original approach, see [23], are used. Note that for this term order we need only a single Gröbner basis computation at this level.

A third version, *zerosolve2!**, mixes the first approach with the FGLM change of term orders. It is not incorporated into the extended Gröbner factorizer.

4.4 Basic Operations on Ideals and Modules

Gröbner and local standard bases are the heart of several basic algorithms in ideal theory, see e.g. [2, 6.2.]. CALI offers the following facilities:

`submodulep!*(m,n)`

tests the dpmat m for being a submodule of the dpmat n reducing the basis elements of m with respect to n. The result will be correct provided n is a Gröbner basis.

`modequalp!*(m,n)`

= submodulep!*(m,n) and submodulep!*(n,m).

`eliminate!*(m,<variable list>)`

computes the elimination ideal/module eliminating the variables in the given variable list (a subset of the variables of the current base ring). Changes temporarily the term order to degrevlex.

`matintersect!* l` [10]

computes the intersection of the dpmats in the dpmat list l along [2, 6.20].

CALI offers several quotient algorithms. They rest on the computation of quotients by a single element of the following kind: Assume $M \subset S^c, v \in S^c, f \in S$. Then there are

the *module quotient* $M : (v) = \{g \in S \mid gv \in M\}$,

the *ideal quotient* $M : (f) = \{w \in S^c \mid fw \in M\}$, and

the *stable quotient* $M : (f)^\infty = \{w \in S^c \mid \exists\, n \,:\, f^n w \in M\}$.

CALI uses the elimination approach [7, 4.4.] and [2, 6.38] for their computation:

`matquot!*(M,f)`

returns the module or ideal quotient $M : (f)$ depending on f.

`matqquot!*(M,f)`

returns the stable quotient $M : (f)^\infty$.

*matquot!** calls the pseudo division with remainder

`dp_pseudodivmod(g,f)`

that returns a dpoly list $\{q, r, z\}$ such that $z \cdot g = q \cdot f + r$ with a dpoly unit z. (g, f and r must belong to the same free module). This is done uniformly for noetherian and local term orders with an extended normal form algorithm as described in [14].

In the same way one defines the quotient of a module by another module (both embedded in a common free module S^c), the quotient of a module by an ideal, and the stable quotient of a module by an ideal. Algorithms for their computation can be obtained from the corresponding algorithms for a single element as divisor either by the generic element method [8] or as an intersection [2, 6.31]. CALI offers both approaches (X=1 or 2 below) at the symbolic level, but for true quotients only the latter one is integrated into the algebraic mode interface.

`idealquotientX!*(M,I)`

returns the ideal quotient $M : I$ of the dpmat M by the dpmat ideal I.

[10] This can be done for ideals and modules in an unique way. Hence *idealintersect!** has been removed in v. 2.1.

`modulequotientX!*(M,N)`

returns the module quotient $M : N$ of the dpmat M by the dpmat N.

`annihilatorX!* M`

returns the annihilator of *coker* M, i.e. the module quotient $S^c : M$, if M is a submodule of S^c.

`matstabquot!*(M,I)`

returns the stable quotient $M : I^\infty$ (only by the general element method).

4.5 Monomial Ideals

Monomial ideals occur as ideals of leading terms of (ideal's) Gröbner bases and also as components of leading term modules of submodules of free modules, see [12], and reflect some properties of the original ideal/module. Several parameters of the original ideal or module may be read off from it as e.g. dimension and Hilbert series.

The *module moid* contains the corresponding algorithms on monomial ideals. Monomial ideals are lists of monomials, kept sorted by descending lexicographic order as proposed in [1].

`moid_primes u`

returns the minimal primes (as a list of lists of variable names) of the monomial ideal u using an adaption of the algorithm, proposed in [1] for the computation of the codimension.

`indepvarsets!* m`

returns (based on *moid_primes*) the list of strongly independent sets of m, see [19] and [12] for definitions.

`dim!* m`

returns the dimension of *coker* m as the size of the largest independent set.

`codim!* m`

returns the codimension of *coker* m.

`easyindepset!* m`

returns a maximal with respect to inclusion independent set of m.

`easydim!* m`

is a fast dimension algorithm (based on *easyindepset*), that will be correct if m is (radically) unmixed. Since it is significantly faster than the general dimension algorithm[11], it should be used, if all maximal independent sets are known to be of equal cardinality (as e.g. for prime or unmixed ideals, see [12]).

[11] This algorithm is of linear time as opposed to the problem to determine the dimension of an arbitrary monomial ideal, that is known to be NP-hard in the number of variables, see [1].

4.6 Hilbert Series

CALI v. 2.2 now offers also *weighted Hilbert series*, i.e. series that may reflect multihomogeneity of ideals and modules. For this purpose a weighted Hilbert series has a list of (integer) degree vectors as second parameter, and the ideal(s) of leading terms are evaluated wrt. these weights. For the output and polynomial arithmetic, involved during the computation of the Hilbert series numerator, the different weight levels are mapped onto the first variables of the current ring. If w is such a weight vector list and I is a monomial ideal in the polynomial ring $S = k[x_v : v \in V]$ we get (using multi exponent notation)

$$H(S/I,t) := \sum_{\alpha} |\{x^a \notin I : w(a) = \alpha\}| \cdot t^{\alpha} = \frac{Q(t)}{\prod_{v \in V}(1 - t^{w(x_v)})}$$

for a certain polynomial Hilbert series numerator $Q(t)$. $H(R/I,t)$ is known to be a rational function with pole order at $t = 1$ equal to $dim\ R/I$. Note that *WeightedHilbertSeries* returns a *reduced* rational function where the gcd of numerator and denominator is canceled out.

(Non weighted) Hilbert series call the weighted Hilbert series procedure with a single weight vector, the ecart vector of the current ring.

The Hilbert series numerator $Q(t)$ is computed using (the obvious generalizations to the weighted case of) the algorithms in [1] and [3]. Experiments suggest that the former is better for few generators of high degree whereas the latter has to be preferred for many generators of low degree. Choose the version with *hftestversion* n, $n = 1, 2$. Bayer/Stillman's approach ($n = 1$) is the default. In the following m is a dpmat and Gröbner basis.

`hf_whilb(m,w)`

returns the weighted Hilbert series numerator $Q(t)$ of m according to the version chosen with *hftestversion*.

`WeightedHilbertSeries!*(m,w)`

returns the weighted Hilbert series reduced rational function of m as s.q.

`HilbertSeries!*(m,w)`

returns the Hilbert series reduced rational function of m wrt. the ecart vector of the current ring as s.q.

`hf_whilb3(u,w)` and `hf_whs_from_resolution(u,w)`

compute the weighted Hilbert series numerator and the corresponding reduced rational function from (the column degrees of) a given resolution u.

`degree!* m`

returns the value of the numerator of the reduced Hilbert series of m at $t = 1$. i.e. the sum of its coefficients. For the standard ecart this is the degree of *coker* m.

4.7 Resolutions

Resolutions of ideals and modules, represented as lists of dpmats, are computed via repeated syzygy computation with minimization steps between them to get minimal bases and generators of syzygy modules. Note that the algorithms apply simultaneously to both Noetherian and non Noetherian term orders. For compatibility reasons with further releases v. 2.2 introduces a second parameter to bound the number of syzygy modules to be computed, since Hilbert's syzygy theorem applies only to regular rings.

`Resolve!*(m,d)`

computes a minimal resolution of the dpmat m, i.e. a list of dpmats $\{s_0, s_1, s_2, \ldots\}$, where s_k is the k-th syzygy module of m, upto part s_d.

`BettiNumbers!* c` and `GradedBettiNumbers!* c`

returns the Betti numbers resp. the graded Betti numbers of the resolution c, i.e. the list of the lengths resp. the degree lists (according to the ecart) themselves of the dpmats in c.

4.8 Zero Dimensional Ideals and Modules

There are several algorithms that either force the reduction of a given problem to dimension zero or work only for zero dimensional ideals or modules. The *module odim* offers such algorithms. It contains, e.g.

`dimzerop!* m`

that tests a dpmat m for being zero dimensional.

`getkbase!* m`

that returns a (monomial) k-vector space basis of $Coker\ m$ provided m is a Gröbner basis.

`odim_borderbasis m`

that returns a border basis, see [20], of the zero dimensional dpmat m as a list of base elements.

`odim_parameter m`

that returns a parameter of the dpmat m, i.e. a variable $x \in vars$ such that $k[x] \bigcap Ann\ S^c/m = (0)$, or *nil* if m is zero dimensional.

`odim_up(a,m)`

that returns an univariate polynomial (of smallest possible degree if m is a gbasis) in the variable a, that belongs to the zero dimensional dpmat ideal m, using Buchberger's approach [5].

4.9 Primary Decomposition and Related Algorithms

The algorithms of the *module prime* implement the ideas of [10] with modifications along [18] and their natural generalizations to modules as e.g. explained in [28]. Version 2.2.1

fixes a serious bug detecting superfluous embedded primary components, see section 1.5, and contains now a second primary decomposition algorithm, based on ideal separation, as standard. For a discussion about embedded primes and the ideal separation approach, see [17].

CALI contains also algorithms for the computation of the unmixed part of a given module and the unmixed radical of a given ideal (along the same lines). We followed the stepwise recursion decreasing dimension in each step by 1 as proposed in (the final version of) [10] rather than the "one step" method described in [2] since handling leading coefficients, i.e. standard forms, depending on several variables is a quite hard job for REDUCE[12].

In the following procedures m must be a Gröbner basis.

`zeroradical!* m`

returns the radical of the zero dimensional ideal m, using squarefree decomposition of univariate polynomials.

`zeroprimes!* m`

computes as in [10] the list of prime ideals of $Ann\ F/M$ if m is zero dimensional, using the (sparse) general position argument from [19].

`zeroprimarydecomposition!* m`

computes the primary components of the zero dimensional dpmat m using prime splitting with the prime ideals of $Ann\ F/M$. It returns a list of pairs with first entry the primary component and second entry the corresponding associated prime ideal.

`isprime!* m`

a (one step) primality test for ideals, extracted from [10].

`isolatedprimes!* m`

computes (only) the isolated prime ideals of $Ann\ F/M$.

`radical!* m`

computes the radical of the dpmat ideal m, reducing as in [10] to the zero dimensional case.

`easyprimarydecomposition!* m`

computes the primary components of the dpmat m, if it has no embedded components. The algorithm uses prime splitting with the isolated prime ideals of $Ann\ F/M$. It returns a list of pairs as in *zeroprimarydecomposition!**.

`primarydecomposition!* m`

computes the primary components of the dpmat m along the lines of [10]. It returns a list of two-element lists as in *zeroprimarydecomposition!**.

[12] *prime!=decompose2* implements this strategy in the symbolic mode layer.

`unmixedradical!* m`

returns the unmixed radical, i.e. the intersection of the isolated primes of top dimension, associated to the dpmat ideal m.

`eqhull!* m`

returns the equidimensional hull, i.e. the intersection of the top dimensional primary components of the dpmat m.

4.10 Advanced Algorithms

The *module scripts* just under further development offers some advanced topics of the Gröbner bases theory. It introduces the new data structure of a *map* between base rings:

A ring map

$$\phi \ : \ R \longrightarrow S$$

for $R = k[r_i], S = k[s_j]$ is represented in symbolic mode as a list

$$\{preimage\_ring\ R,\ image\_ring\ S, subst\_list\},$$

where `subst_list` is a substitution list $\{r_1 = \phi_1(s), r_2 = \phi_2(s), \ldots\}$ in algebraic prefix form, i.e. looks like `(list (equal var image) ...)`.

The central tool for several applications is the computation of the preimage $\phi^{-1}(I) \subset R$ of an ideal $I \subset S$ either under a polynomial map ϕ or its closure in R under a rational map ϕ, see [2, 7.69 and 7.71].

`preimage!*(m,map)`

computes the preimage of the ideal m in algebraic prefix form under the given polynomial map and sets the current base ring to the preimage ring. Returns the result also in algebraic prefix form.

`ratpreimage!*(m,map)`

computes the closure of the preimage of the ideal m in algebraic prefix form under the given rational map and sets the current base ring to the preimage ring. Returns the result also in algebraic prefix form.

Derived applications are

`affine_monomial_curve!*(l,vars)`

l is a list of integers, *vars* a list of variable names of the same length as l. The procedure sets the current base ring and returns the defining ideal of the affine monomial curve with generic point $(t^i \ : \ i \in l)$ computing the corresponding preimage.

`analytic_spread!* M`

Computes the analytic spread of M, i.e. the dimension of the exceptional fiber $\mathcal{R}(M)/m\mathcal{R}(M)$ of the blowup along M over the irrelevant ideal m of the current base ring.

`assgrad!*(M,N,vars)`

Computes the associated graded ring

$$gr_R(N) := (R/N \oplus N/N^2 \oplus \ldots) = \mathcal{R}(N)/N\mathcal{R}(N)$$

over the ring $R = S/M$, where M and N are dpmat ideals defined over the current base ring S. `vars` is a list of new variable names one for each generator of N. They are used to create a second ring T with degree order corresponding to the ecart of the row degrees of N and a ring map

$$\phi : S \oplus T \longrightarrow S.$$

It returns a dpmat ideal J such that $(S \oplus T)/J$ is a presentation of the desired associated graded ring over the new current base ring $S \oplus T$.

`blowup!*(M,N,vars)`

Computes the blow up $\mathcal{R}(N) := R[N \cdot t]$ of N over the ring $R = S/M$, where M and N are dpmat ideals defined over the current base ring S. `vars` is a list of new variable names one for each generator of N. They are used to create a second ring T with degree order corresponding to the ecart of the row degrees of N and a ring map

$$\phi : S \oplus T \longrightarrow S.$$

It returns a dpmat ideal J such that $(S \oplus T)/J$ is a presentation of the desired blowup ring over the new current base ring $S \oplus T$.

`proj_monomial_curve!*(l,vars)`

l is a list of integers, *vars* a list of variable names of the same length as l. The procedure set the current base ring and returns the defining ideal of the projective monomial curve with generic point $(s^{d-i} \cdot t^i \ : \ i \in l)$ in R, where $d = max\{x \ : \ x \in l\}$, computing the corresponding preimage.

`sym!*(M,vars)`

Computes the symmetric algebra $Sym(M)$ where M is a dpmat ideal defined over the current base ring S. `vars` is a list of new variable names one for each generator of M. They are used to create a second ring R with degree order corresponding to the ecart of the row degrees of N and a ring map

$$\phi : S \oplus R \longrightarrow S.$$

It returns a dpmat ideal J such that $(S \oplus R)/J$ is the desired symmetric algebra over the new current base ring $S \oplus R$.

There are several other applications:

`minimal_generators!* m`

returns a set of minimal generators of the dpmat m inspecting the first syzygy module.

`nzdp!*(f,m)`

tests whether the dpoly f is a non zero divisor on *coker* m. m must be a Gröbner basis.

`symbolic_power!*(m,d)`

returns the dth symbolic power of the prime dpmat ideal m as the equidimensional hull of the dth true power. (Hence applies also to unmixed ideals.)

`varopt!* m`

finds a heuristically optimal variable order by the approach in [4] and returns the corresponding list of variables.

4.11 Dual Bases

For the general ideas underlying the dual bases approach see e.g. [20]. This paper explains, that constructive problems from very different areas of commutative algebra can be formulated in a unified way as the computation of a basis for the intersection of the kernels of a finite number of linear functionals generating a dual S-module. Our implementation honours this point of view, presenting two general drivers *dualbases* and *dualhbases* for the computation of such bases (even as submodules of a free module $M = S^m$) with affine resp. projective dimension zero.

Such a collection of N linear functionals

$$L : M = S^m \longrightarrow k^N$$

should be given through values $\{[e_i, L(e_i)], i = 1, \ldots, m\}$ on the generators e_i of M and an evaluation function `evlf([p,L(p)],x)`, that evaluates $L(p \cdot x)$ from $L(p)$ for $p \in M$ and the variable $x \in S$.

dualbases starts with a list of such generator/value constructs generating M and performs Gaussian reduction on expressions $[p \cdot x, L(p \cdot x)]$, where p was already processed, $L(p) \neq 0$, and $x \in S$ is a variable. These elements are processed in ascending order wrt. the term order on M. This guarantees both termination and that the resulting basis of *ker* L is a Gröbner basis. The N values of L are attached to N variables, that are ordered linearly. Gaussian elimination is executed wrt. this variable order.

To initialize the dual bases driver one has to supply the basic generator/value list (through the parameter list; for ideals just the one element list containing the generator $[1 \in S, L(1)]$), the evaluation function, and the linear algebra variable order. The latter are supplied via the property list of `cali` as properties `evlf` and `varlessp`. Different applications need more entries on the property list of `cali` to manage the communication between the driver and the calling routine.

dualhbases realizes the same idea for (homogeneous) ideals and modules of (projective) dimension zero. It produces zerodimensional "slices" with ascending degree until it reaches a supremum supplied by the user, see [20] for details.

Applications concern affine and projective defining ideals of a finite number of points[13] and two versions (with and without precomputed border basis) of term order changes for zerodimensional ideals and modules as first described in [9].

`affine_points!* m`

> m is a matrix of domain elements (in algebraic prefix form) with as many columns as the current base ring has ring variables. This procedure returns the defining ideal of the collection of points in affine space with coordinates given by the rows of m. Note that m may contain parameters. In this case k is treated as rational function field.

`change_termorder!*(m,r)` and `change_termorder1!*(m,r)`

> m is a Gröbner basis of a zero dimensional ideal wrt. the current base ring. These procedures change the current ring to r and compute the Gröbner basis of m wrt. the new ring r. The former uses a precomputed border basis.

`proj_points!* m`

> m is a matrix of domain elements (in algebraic prefix form) with as many columns as the current base ring has ring variables. This procedure returns the defining ideal of the collection of points in projective space with homogeneous coordinates given by the rows of m. Note that m may as for `affine_points` contain parameters.

[13] This substitutes the "brute force" method computing the corresponding intersections directly as it was implemented in v. 2.1. The new approach is significantly faster. The old stuff is available as *affine_points1!** and *proj_points1!**.

A A Short Description of Procedures Available in Algebraic Mode

Here we give a short description, ordered alphabetically, of **algebraic** procedures offered by CALI in the algebraic mode interface[14].

If not stated explicitely procedures take (algebraic mode) polynomial matrices ($c > 0$) or polynomial lists ($c = 0$) $m, m1, m2, \ldots$ as input and return results of the same type. *gb* stands for a bounded identifier[15], *gbr* for one with precomputed resolution. For the mechanism of *bounded identifier* see the section "Algebraic Mode Interface".

`affine_monomial_curve(l,vars)`

l is a list of integers, *vars* a list of variable names of the same length as l. The procedure sets the current base ring and returns the defining ideal of the affine monomial curve with generic point $(t^i \ : \ i \in l)$.

`affine_points m`

m is a matrix of domain elements (in algebraic prefix form) with as many columns as the current base ring has ring variables. This procedure returns the defining ideal of the collection of points in affine space with coordinates given by the rows of m. Note that m may contain parameters. In this case k is treated as rational function field.

`analytic_spread m`

Computes the analytic spread of m.

`annihilator m`

returns the annihilator of the dpmat $m \subseteq S^c$, i.e. $Ann\ S^c/M$.

`assgrad(M,N,vars)`

Computes the associated graded ring $gr_R(N)$ over $R = S/M$, where S is the current base ring. `vars` is a list of new variable names, one for each generator of N. They are used to create a second ring T to return an ideal J such that $(S \oplus T)/J$ is the desired associated graded ring over the new current base ring $S \oplus T$.

`bettiNumbers gbr`

extracts the list of Betti numbers from the resolution of *gbr*.

`blowup(M,N,vars)`

Computes the blow up $\mathcal{R}(N)$ of N over the ring $R = S/M$, where S is the current base ring. `vars` is a list of new variable names, one for each generator of N. They are used to create a second ring T to return an ideal J such that $(S \oplus T)/J$ is the desired blowup ring over the new current base ring $S \oplus T$.

[14] It does **not** contain switches, get... procedures, setting trace level and related stuff.

[15] Different to v. 2.1 a Gröbner basis will be computed automatically, if necessary.

`change_termorder(m,r)` and `change_termorder1(m,r)`

Change the current ring to r and compute the Gröbner basis of m wrt. the new ring r by the FGLM approach. The former uses internally a precomputed border basis.

`codim gb`

returns the codimension of S^c/gb.

`degree gb`

returns the multiplicity of gb as the sum of the coefficients of the (classical) Hilbert series numerator.

`degsfromresolution gbr`

returns the list of column degrees from the minimal resolution of gbr.

`deleteunits m`

factors each basis element of the dpmat ideal m and removes factors that are polynomial units. Applies only to non Noetherian term orders.

`dim gb`

returns the dimension of S^c/gb.

`dimzerop gb`

tests whether S^c/gb is zerodimensional.

`directsum(m1,m2,...)`

returns the direct sum of the modules $m1, m2, \ldots$, embedded into the direct sum of the corresponding free modules.

`dpgcd(f,g)`

returns the gcd of two polynomials f and g, computed by the syzygy method.

`easydim m` and `easyindepset m`

If the given ideal or module is unmixed (e.g. prime) then all maximal strongly independent sets are of equal size and one can look for a maximal with respect to inclusion rather than size strongly independent set. These procedures don't test the input for being a Gröbner basis or unmixed, but construct a maximal with respect to inclusion independent set of the basic leading terms resp. detect from this (an approximation for) the dimension.

`easyprimarydecomposition m`

a short primary decomposition using ideal separation of isolated primes of m, that yields true results only for modules without embedded components. Returns a list of $\{component, associated\ prime\}$ pairs.

`eliminate(m,<variable list>)`

computes the elimination ideal/module eliminating the variables in the given variable list (a subset of the variables of the current base ring). Changes temporarily the term order to degrevlex.

`eqhull m`

returns the equidimensional hull of the dpmat m.

`extendedgroebfactor(m,c)` and `extendedgroebfactor1(m,c)`

return for a polynomial ideal m and a list of (polynomial) constraints c a list of results $\{b_i, c_i, v_i\}$, where b_i is a triangular set wrt. the variables v_i and c_i is a list of constraints, such that $Z(m,c) = \bigcup Z(b_i, c_i)$. For the first version the output may be not minimal. The second version decides superfluous components already during the algorithm.

`gbasis gb`

returns the Gröbner resp. local standard basis of gb.

`getkbase gb`

returns a k-vector space basis of S^c/gb, consisting of module terms, provided gb is zerodimensional.

`getleadterms gb`

returns the dpmat of leading terms of a Gröbner resp. local standard basis of gb.

`GradedBettinumbers gbr`

extracts the list of degree lists of the free summands in a minimal resolution of gbr.

`groebfactor(m[,c])`

returns for the dpmat ideal m and an optional constraint list c a (reduced) list of dpmats such that the union of their zeroes is exactly $Z(m,c)$. Factors all polynomials involved in the Gröbner algorithms of the partial results.

`HilbertSeries gb`

returns the Hilbert series of gb with respect to the current ecart vector.

`homstbasis m`

computes the standard basis of m by Lazard's homogenization approach.

`ideal2mat m`

converts the ideal (=list of polynomials) m into a column vector.

`ideal_of_minors(mat,k)`

computes the generators for the ideal of k-minors of the matrix mat.

`ideal_of_pfaffians(mat,k)`

computes the generators for the ideal of the $2k$-pfaffians of the skewsymmetric matrix mat.

`idealpower(m,n)`

returns the interreduced basis of the ideal power m^n with respect to the integer $n \geq 0$.

`idealprod(m1,m2,...)`

returns the interreduced basis of the ideal product $m1 \cdot m2 \cdot \ldots$ of the ideals $m1, m2, \ldots$.

`idealquotient(m1,m2)`

returns the ideal quotient $m1 : m2$ of the module $m1 \subseteq S^c$ by the ideal $m2$.

`idealsum(m1,m2,...)`

returns the interreduced basis of the ideal sum $m1 + m2 + \ldots$.

`indepvarsets gb`

returns the list of strongly independent sets of gb with respect to the current term order, see [19] for a definition in the case of ideals and [12] for submodules of free modules.

`initmat(m,<file name>`

initializes the dpmat m together with its base ring, term order and column degrees from a file.

`interreduce m`

returns the interreduced module basis given by the rows of m, i.e. a basis with pairwise indivisible leading terms.

`isolatedprimes m`

returns the list of isolated primes of the dpmat m, i.e. the isolated primes of $Ann\ S^c/M$.

`isprime gb`

tests the ideal gb to be prime.

`iszeroradical gb`

tests the zerodimensional ideal gb to be radical.

`lazystbasis m`

computes the standard basis of m by the lazy algorithm, see e.g. [26].

`listgroebfactor in`

computes for the list in of ideal bases a list out of Gröbner bases by the Gröbner factorization method, such that $\bigcup_{m \in in} Z(m) = \bigcup_{m \in out} Z(m)$.

`mat2list m`

converts the matrix m into a list of its entries.

`matappend(m1,m2,...)`

collects the rows of the dpmats $m1, m2, \ldots$ to a common matrix. $m1, m2, \ldots$ must be submodules of the same free module, i.e. have equal column degrees (and size).

`mathomogenize(m,var)` [16]

returns the result obtained by homogenization of the rows of m with respect to the variable `var` and the current *ecart vector*.

`matintersect(m1,m2,...)`

returns the interreduced basis of the intersection $m1 \bigcap m2 \bigcap \ldots$.

`matjac(m,<variable list>)`

returns the Jacobian matrix of the ideal m with respect to the supplied variable list

`matqquot(m,f)`

returns the stable quotient $m : (f)^\infty$ of the dpmat m by the polynomial $f \in S$.

`matquot(m,f)`

returns the quotient $m : (f)$ of the dpmat m by the polynomial $f \in S$.

`matstabquot(m1,id)`

returns the stable quotient $m1 : id^\infty$ of the dpmat $m1$ by the ideal id.

`matsum(m1,m2,...)`

returns the interreduced basis of the module sum $m1 + m2 + \ldots$ in a common free module.

`minimal_generators m`

returns a set of minimal generators of the dpmat m.

`minors(m,b)`

returns the matrix of minors of size $b \times b$ of the matrix m.

`a mod m`

computes the (true) normal form(s), i.e. a standard quotient representation, of a modulo the dpmat m. a may be either a polynomial or a polynomial list ($c = 0$) or a matrix ($c > 0$) of the correct number of columns.

`modequalp(gb1,gb2)`

tests, whether $gb1$ and $gb2$ are equal (returns YES or NO).

`modulequotient(m1,m2)`

returns the module quotient $m1 : m2$ of two dpmats $m1, m2$ in a common free module.

[16] Dehomogenize with `sub(z=1,m)` if z is the homogenizing variable.

`normalform(m1,m2)`

returns a list of three dpmats $\{m3, r, z\}$, where $m3$ is the normalform of $m1$ modulo $m2$, z a scalar matrix of polynomial units (i.e. polynomials of degree 0 in the noetherian case and polynomials with leading term of degree 0 in the tangent cone case), and r the relation matrix, such that

$$m3 = z * m1 + r * m2.$$

`nzdp(f,m)`

tests whether the dpoly f is a non zero divisor on *coker* m.

`pfaffian mat`

returns the pfaffian of a skewsymmetric matrix mat.

`preimage(m,map)`

computes the preimage of the ideal m under the given polynomial map and sets the current base ring to the preimage ring.

`primarydecomposition m`

returns the primary decomposition of the dpmat m as a list of $\{component, associated\ prime\}$ pairs.

`proj_monomial_curve(l,vars)`

l is a list of integers, $vars$ a list of variable names of the same length as l. The procedure sets the current base ring and returns the defining ideal of the projective monomial curve with generic point $(s^{d-i} \cdot t^i \ : \ i \in l)$ in R where $d = max\{x \ : \ x \in l\}$.

`proj_points m`

m is a matrix of domain elements (in algebraic prefix form) with as many columns as the current base ring has ring variables. This procedure returns the defining ideal of the collection of points in projective space with homogeneous coordinates given by the rows of m. Note that m may as for `affine_points` contain parameters.

`radical m`

returns the radical of the dpmat ideal m.

`random_linear_form(vars,bound)`

returns a random linear form in the variables `vars` with integer coefficients less than the supplied `bound`.

`ratpreimage(m,map)`

computes the closure of the preimage of the ideal m under the given rational map and sets the current base ring to the preimage ring.

`resolve(m[,d])`

returns the first d members of the minimal resolution of the bounded identifier m as a list of matrices. If the resolution has less than d non zero members, only those are collected. (Default: $d = 100$)

`savemat(m,<file name>)`

save the dpmat m together with the settings of it base ring, term order and column degrees to a file.

`setgbasis m`

declares the rows of the bounded identifier m to be already a Gröbner resp. local standard basis thus avoiding a possibly time consuming Gröbner or standard basis computation.

`sieve(m,<variable list>)`

sieves out all base elements with leading terms having a factor contained in the specified variable list (a subset of the variables of the current base ring). Useful for elimination problems solved "by hand".

`singular_locus(M,c)`

returns the defining ideal of the singular locus of $Spec\ S/M$ where M is an ideal of codimension c, adding to M the generators of the ideal of the c-minors of the Jacobian of M.

`submodulep(m,gb)`

tests, whether m is a submodule of gb (returns YES or NO).

`sym(M,vars)`

Computes the symmetric algebra $Sym(M)$ where M is an ideal defined over the current base ring S. `vars` is a list of new variable names, one for each generator of M. They are used to create a second ring R to return an ideal J such that $(S \oplus R)/J$ is the desired symmetric algebra over the new current base ring $S \oplus R$.

`symbolic_power(m,d)`

returns the dth symbolic power of the prime dpmat ideal m.

`syzygies m`

returns the first syzygy module of the bounded identifier m.

`tangentcone gb`

returns the tangent cone part, i.e. the homogeneous part of highest degree with respect to the first degree vector of the term order from the Gröbner basis elements of the dpmat gb. The term order must be a degree order.

`unmixedradical m`

returns the unmixed radical of the dpmat ideal m.

`varopt m`

finds a heuristically optimal variable order, see [4].

$$\texttt{vars} := \texttt{varopt m};\ \ \texttt{setring}(\texttt{vars}, \{\}, \texttt{lex});\ \ \texttt{setideal}(\texttt{m}, \texttt{m});$$

changes to the lexicographic term order with heuristically best performance for a lexicographic Gröbner basis computation.

`WeightedHilbertSeries(m,w)`

returns the weighted Hilbert series of the dpmat m. Note that m is not a bounded identifier and hence not checked to be a Gröbner basis. w is a list of integer weight vectors.

`zeroprimarydecomposition m`

returns the primary decomposition of the zerodimensional dpmat m as a list of $\{component, associated\ prime\}$ pairs.

`zeroprimes m`

returns the list of primes of the zerodimensional dpmat m.

`zeroradical gb`

returns the radical of the zerodimensional ideal gb.

`zerosolve m`, `zerosolve1 m` and `zerosolve2 m`

Returns for a zerodimensional ideal a list of triangular systems that cover $Z(m)$. `Zerosolve` needs a pure lex. term order for the "fast" turn to lex. as described in [23], `Zerosolve1` is the "slow" turn to lex. as described in [16], and `Zerosolve2` incorporated the FGLM term order change into `Zerosolve1`.

B The CALI Module Structure

name	subject	data type	representation
cali	Header module, contains global variables, switches etc.	—	—
bcsf	Base coefficient arithmetic	base coeff.	standard forms
ring	Base ring setting, definition of the term order	base ring	special type RING
mo	monomial arithmetic	monomials	(exp. list . degree list)
dpoly	Polynomial and vector arithmetic	dpolys	list of terms
bas	Operations on base lists	base list	list of base elements
dpmat	Operations on polynomial matrices, the central data type of CALI	dpmat	special type DPMAT
red	Normal form algorithms	—	—
groeb	Gröbner basis algorithm and related ones	—	—
groebf	the Gröbner factorizer and its extensions	—	—
matop	Operations on (lists of) dpmats that correspond to ideal/module operations	—	—
quot	Different quotient algorithms	—	—
moid	Monomial ideal algorithms	monomial ideal	list of monomials
hf	weighted Hilbert series	–	–
res	Resolutions of dpmats	resolution	list of dpmats
intf	Interface to algebraic mode	—	—
odim	Algorithms for zerodimensional ideals and modules	—	—
prime	Primary decomposition and related questions	—	—
scripts	Advanced applications	—	—
calimat	Extension of the matrix package	—	—
lf	The dual bases approach	—	—
triang	(Zero dimensional) triangular systems	—	—

Index

References

[1] D. Bayer, M. Stillman: Computation of Hilbert functions. *J. Symb. Comp.* **14** (1992), 31 - 50.

[2] T. Becker, H. Kredel, V. Weispfenning: Gröbner bases. A computational approach to commutative algebra. Grad. Texts in Math. 141, Springer, New York 1993.

[3] A. M. Bigatti, P. Conti, L. Robbiano, C. Traverso: A "divide and conquer" algorithm for Hilbert-Poincare series, multiplicity and dimension of monomial ideals. In: Proc. AAECC-10, LNCS 673 (1993), 76 - 88.

[4] W. Boege, R. Gebauer, H. Kredel: Some examples for solving systems of algebraic equations by calculating Gröbner bases. *J. Symb. Comp.* **2** (1986), 83 - 98.

[5] B. Buchberger: Gröbner bases: An algorithmic method in polynomial ideal theory. In: Recent trends in multidimensional system theory (N. K. Bose ed), Reidel, Dortrecht 1985, 184 - 232.

[6] B. Buchberger: Applications of Gröbner bases in non-linear computational geometry. LNCS 296 (1988), 52 - 80.

[7] D. Cox, J. Little, D. O'Shea: Ideals, varieties, and algorithms. Undergraduate Texts in Math., Springer, New York 1992.

[8] D. Eisenbud: Commutative algebra with a view toward algebraic geometry. Springer, 1995.

[9] Faugere, Gianni, Lazard, Mora: Efficient computations of zerodimensional Gröbner bases by change of ordering. *J. Symb. Comp.* **16** (1993), 329 - 344.

[10] P. Gianni, B. Trager, G. Zacharias: Gröbner bases and primary decomposition of polynomial ideals. *J. Symb. Comp.* **6** (1988), 149 - 167.

[11] A. Giovini, T. Mora, G. Niesi, L. Robbiano, C. Traverso: "One sugar cube, please" or Selection strategies in the Buchberger algorithm. In: Proceedings of the ISSAC'91, ACM Press 1991, 49 - 54.

[12] H.-G. Gräbe: Two remarks on independent sets. *J. Alg. Comb.* **2** (1993), 137 - 145.

[13] H.-G. Gräbe: The tangent cone algorithm and homogenization. *J. Pure Applied Alg.***97** (1994), 303 - 312.

[14] H.-G. Gräbe: Algorithms in local algebra. To appear

[15] H.-G. Gräbe: On factorized Gröbner bases. Report Nr. 6 (1994), Inst. f. Informatik, Univ. Leipzig.

To appear in: Proc. "Computer Algebra in Science and Engineering", Bielefeld 1994.

[16] H.-G. Gräbe: Triangular systems and factorized Gröbner bases. Report Nr. 7 (1995), Inst. f. Informatik, Univ. Leipzig.

[17] H.-G. Gräbe: Factorized Gröbner bases and primary decomposition. To appear.

[18] H. Kredel: Primary ideal decomposition. In: Proc. EUROCAL'87, Lecture Notes in Comp. Sci. 378 (1986), 270 - 281.

[19] H. Kredel, V. Weispfenning: Computing dimension and independent sets for polynomial ideals. *J. Symb. Comp.* **6** (1988), 231 - 247.

[20] M. Marinari, H.-M. Möller, T. Mora: Gröbner bases of ideals given by dual bases. In: Proc. ISSAC'91, ACM Press 1991, 55 - 63.

[21] B. Mishra: Algorithmic Algebra. Springer, New York 1993.

[22] H.-M. Möller, F. Mora: New constructive methods in classical ideal theory. *J. Alg.* **100** (1986), 138 -178.

[23] H.-M. Möller: On decomposing systems of polynomial equations with finitely many solutions. *J. AAECC* **4** (1993), 217 - 230.

[24] T. Mora, L. Robbiano: The Gröbner fan of an ideal. *J. Symb. Comp.* **6** (1988), 183 - 208.

[25] T. Mora: Seven variations on standard bases. Preprint, Univ. Genova, 1988.

[26] T. Mora, G. Pfister, C. Traverso: An introduction to the tangent cone algorithm. In: *Issues in non-linear geometry and robotics, C.M. Hoffman ed.*, JAI Press.

[27] L. Robbiano: Computer algebra and commutative algebra. LNCS 357 (1989), 31 - 44.

[28] E. W. Rutman: Gröbner bases and primary decomposition of modules. *J. Symb. Comp.* **14** (1992), 483 - 503.