GBNP

computing Gröbner bases of noncommutative polynomials

1.1.0

29 August 2024

A.M. Cohen

J.W. Knopper

A.M. Cohen
Email: A.M.Cohen@tue.nl

J.W. Knopper
Email: J.W.Knopper@tue.nl

Address: TU/e,
POB 513, 5600 MB Eindhoven, the Netherlands

Abstract

We provide algorithms, written in the GAP 4 programming language, for computing Gröbner bases of non-commutative polynomials, and some variations, such as a weighted and truncated version and a tracing facility. In addition, there are algorithms for analyzing the quotient of a non-commutative polynomial algebra by a 2-sided ideal generated by a set of polynomials whose Gröbner basis has been determined and for computing quotient modules of free modules over quotient algebras.

The notion of algorithm is interpreted loosely: in general one cannot expect a non-commutative Gröbner basis algorithm to terminate, as it would imply solvability of the word problem for finitely presented (semi)groups.

This documentation gives a short description of the mathematical content, explains the functions of the package, and provides more than twenty worked out examples.

Copyright

Acknowledgements

- The package is based on an earlier version by Rosane Ushirobira.
- The bulk of the package is written by Arjeh M. Cohen and Dié A.H. Gijsbers.
- The theory is mainly taken from literature by Teo Mora [Mor94] and Edward L. Green [Gre99].
- From Version 0.8.3 on the package has three additional files (`fincheck.g`, `tree.g` `graphs.g`) with routines for finding the Hilbert function and testing finite dimensionality when given a Gröbner basis by Chris Krook [Kro03], based on work by Victor Ufnarovski [Ufn89].
- From Version 0.9 on the package is enriched with support for fields implemented in GAP and additional prefix rules for quotient modules, as well as some speed improvements by Jan Willem Knopper. Knopper has also formatted the documentation in GAPDoc [LN06].
- From Version 1.0 on the package is extended with NMO (for Noncommutative Monomial Orderings) by Randall Cone. This enables the GBNP user to choose a wider selection of monomial orderings than the standard one built into GBNP itself. Documentation on NMO can be found in the NMO manual [Con10].

Contents

Chapter 1

Introduction

This package, named GBNP for Gröbner Bases for Non-commutative Polynomials, is intended for computing in (associative) non-commutative algebras with a finite presentation. Starting from a free algebra A on a finite number of generating variables, the reader can specify a finite set G of polynomials in these variables, in order to study the quotient algebra of A by the (2-sided) ideal of A generated by G.

This documentation gives a short description of the mathematical content in Chapter 2, explains the functions of the package in Chapter 3, and provides more than twenty four worked out examples in Appendix A. It is available as an HTML document at https://gap-packages.github.io/gbnp/doc/chap0.html.

1.1 Installation

To install GBNP, first download GBNP-1.1.0.tar.gz from https://gap-packages.github.io/gbnp/, then unpack GBNP-1.1.0.tar.gz in the pkg subdirectory of your GAP installation (or in the pkg subdirectory of any other GAP root directory, for example one added with the -l argument) with the following command: tar -xvzf GBNP-1.1.0.tar.gz.

GBNP is then loaded with the GAP command

```
gap> LoadPackage( "GBNP" );
```

1.2 Using the package

If you wish to compute a Gröbner basis, create a list of NPs (non-commutative polynomials in NP format), as described in Section 2.1. This can be done either directly or by use of the transition functions described in Section 3.1. To run the standard algorithm use the functions from Section 3.4. With these functions, you can try and find a Gröbner basis. The word try is included because the algorithm for computing Gröbner bases is not guaranteed to terminate. Printing issues for polynomials in NP format are discussed in Section 3.2. If the Gröbner basis is found and the dimension of the quotient algebra Q (see Section 2.9) is finite, you can find a basis of monomials for Q with the functions in Section 3.5. For a more advanced analysis of Q, such as a proof of finite or infinite dimensionality, or for determining its growth or its partial Hilbert series, use the functions from Section 3.6 .

There are three variants of the Gröbner basis algorithm, the truncated version, the trace version, and the module version. In the (weighted) homogeneous case (described in Section 2.6), the truncated version, given by the functions described in Section 3.8, computes the part of a Gröbner basis up to an indicated weight. The trace version (described in Section 2.5), given by the functions described in Section 3.7, computes an expression of the polynomials of the Gröbner basis found in terms of the original generators. The module version (described in Sections 2.2, 2.7, and 2.8), given by the functions described in Section 3.9, computes a Gröbner basis for a submodule of a free Q-module of finite rank.

Read the example files in Chapter A for inspiration. The source of the files can be perused for auxiliary functions, which are often used in the main functions but not deemed necessary for a first time user.

1.3 Further documentation

The reports [Coh07], [Kro03], and [Kno04] can be downloaded from the web at these addresses:

The report "Non-commutative polynomial computations", by Arjeh M. Cohen (with support of Dié Gijsbers, Jan Willem Knopper, and Chris Krook) can be downloaded from `http://mathdox.org/products/gbnp/gbnp.pdf`.

The report "Dimensionality of quotient algebras", by Chris Krook can be downloaded from `http://mathdox.org/products/gbnp/dqa.pdf`.

The report "GBNP and vector enumeration", by Jan Willem Knopper can be downloaded from `http://mathdox.org/products/gbnp/knopper.pdf`.

Chapter 2

Description

2.1 Non-commutative Polynomials (NPs)

The main datatype of the GBNP package is a list of non-commutative polynomials (NPs). The data type for a *non-commutative polynomial*, referred to as its NP format, is a list of two lists:

- The first list is a list `m` of monomials.
- The second list is a list `c` of coefficients of these monomials.

The two lists have the same length. The polynomial represented by the ordered pair `[m,c]` is $\sum_i c_i m_i$.

A monomial is a list of positive integers. They are interpreted as the indices of the variables. So, if `k = [1,2,3,2,1]` and the variables are x,y,z (in this order), then `k` represents the monomial $xyzyx$. By the way, the name of the variables has no meaning. There are various ways to print these but the default is a,b,c,... (see below).

The zero polynomial is represented by `[[],[]]`. The polynomial 1 is represented by `[[[]],[1]]`.

The algorithms work for the algebra $\mathbb{F}\langle\langle x_1, x_2, \ldots, x_t\rangle\rangle$ of non-commutative polynomials in `t` variables over the field $\mathbb{F}$. Accordingly, the list `c` should contain elements of $\mathbb{F}$. It is not always easy to recover $\mathbb{F}$ from the list `c`. The GAP functions `One` and `Zero` can be of some help.

In order to facilitate viewing the polynomials, we provide the function `PrintNP` (3.2.1). For instance

```
PrintNP([[[1,2],[2,1]],[3,-1]]);
```

yields

```
3ab - ba
```

Indeed, we have the names: `a`, `b`, `c`, ... for x_1, x_2, x_3, ..., except that everything beyond l (the 12-th letter) is called x. This can be easily changed by calling the function `GBNP.ConfigPrint`, which can be found in Section 3.2.

The function `PrintNPList` (3.2.3) is available for printing a list of NPs (=non-commutative polynomials).

In order to facilitate testing whether two data structures represent the same NP, we use the convention that polynomials are "clean". This means that they look as if they are output of the function `CleanNP` (3.3.7). In other words:

- each monomial occurs at most once in the list of monomials,
- no monomials occur whose coefficients are zero,
- the monomials are ordered (total degree first, then lexicographically) from big to small.

An advantage of the ordering is that the leading monomial of an NP `p` is just `p[1][1]` and that its leading coefficient is `p[2][1]`. Users who want to work with other orderings can use the functions defined in the NMO extension [Con10] to GBNP.

2.2 Non-commutative Polynomials for Modules (NPMs)

In Section 2.1 the NP format for elements of a free algebra A of non-commutative polynomials in a fixed number of variables is described. This format can be adjusted slightly to allow the use of a free right module A^n of finite rank n over A. The internal format of an element of the module is similar to that of a non-commutative polynomial. The only change is that each monomial will start with a negative number. The absolute value of this number is the index of the standard basis vector of the free module.

For example in the free $\mathbb{F}\langle\langle x_1, x_2, \ldots, x_t\rangle\rangle$-module of rank 3, the expression `[[[-1]],[1]]` represents $[1,0,0]$ and `[[[-1,1,2],[-1,2,1],[-3,2,2,2]],[6,-7,9]]` represents $[6x_1x_2 - 7x_2x_1, 0, 9x_2^3]$. The zero vector is the represented in the same way as its NP format counterpart in 2.1 and the only one without a negative entry: `[[],[]]`. We refer to this format as the NPM format.

Elements of modules are printed as vectors. See Section 3.9 on how to use modules. Examples A.19, A.21, and A.20 are also recommended.

2.3 Core functions

The core function is `SGrobner` (3.4.2) (which is short for Strong Gröbner, as we use the Strong Normal Form, discussed in Section `StrongNormalFormNPM` (3.9.5), most of the time). It takes a list of NPs in a free algebra A and prepares two lists for treatment in a loop:

- First the list itself, called `G`. Before entering the loop, `G` is cleaned, ordered, and its elements are made monic, that is, multiplied by a scalar so that the leading coefficient becomes one. The ordering is done by comparison of leading monomials. The ordering on leading monomials is length lexicographic. For other orderings, the functions of the NMO extension can be used; see [Con10].
- Second the list of all normal forms with respect to `G` of S-polynomials of elements of `G`. This list is called `D`. For a Gröbner basis, the S-polynomials of polynomials in `D` (possibly with an element of `G`) need to be computed. If `D` is empty then `G` is a Gröbner basis.

Then, the function calls the routine `GBNP.SGrobnerLoop` on the arguments `G`, `D` which are changed in an attempt to modify `G` while still preserving the following two properties.

1. `G` generates the same two-sided ideal I in A as before.
2. `D` contains all normal forms with respect to `G` of S-polynomials of elements from `G` that need to reduce to zero for the basis to be a Gröbner basis.

The importance of this feature is that, in case of huge computations, the user may store G and D at almost any time and resume the computation by reloading G and D and calling the loop function GBNP.SGrobnerLoop whenever convenient. The only technical detail to handle is that the last element of the list G should be copied into the D list. The loop itself performs a step towards making G more like a Gröbner basis of I. As in the commutative case, the progress can be indicated by use of an ordering on the set of leading monomials of the elements of G.

In contrast to the commutative case, however, this ordering is not well founded, and there is no a priori guarantee that the loop will be exited after a finite number of iterations. The loop ends when the list D is empty, in which case the work is essentially done: after some internal cleaning and a bit of further rewriting, the computation is over.

There is also a Grobner (3.4.1) function. It uses (at some places) the Normal Form instead of the Strong Normal Form algorithm. In most of our applications, this usually led to slower performance, so we are not very keen to use it.

In many of our own applications, the full polynomial ring modulo the two-sided ideal I generated by G is a finite-dimensional quotient algebra. In such cases, one would like to know the dimension (whence the function DimQA (3.5.2), QA for Quotient Algebra), find a basis (whence the function BaseQA (3.5.1)), or just the monomials up to a certain degree that are not divisible by a leading term of G (whence the function GBNP.NondivMons). Actually by use of MulQA (3.5.5), you can even multiply elements of the quotient algebra. In case it is unknown whether the quotient algebra is finite or infinite, one can use the functions FinCheckQA (3.6.2) and DetermineGrowthQA (3.6.1). When the quotient algebra is infinite dimensional you may want to determine its partial Hilbert Series. This can be done with the function HilbertSeriesQA (3.6.3).

2.4 About the implementation

Rather than storing all obstructions, the Gröbner basis algorithm computes the (Strong) Normal Form of obstructions from G and puts them into D whenever nonzero. At the beginning of the loop, we take the first element of the D list and prepare it for addition to G. We are then concerned with two goals:

1. to restore the invariant properties,
2. to clean up G (that is, reduce it to a more succinct, shorter set).

This is mainly done by means of additional S-polynomial and Normal Form computations.

As for data management, we have chosen to work with lists in situ, that is, not to copy the list but rather perform all operations on one and the same list. To this end we use operations like RemoveElmList and Add, see **Reference: Add**. The idea here is to economize on space for large computations. We do not use in situ operations everywhere, but have concentrated on the potentially biggest lists: G and D.

For checking whether a monomial can be reduced, an internal tree structure is used.

2.5 Tracing variant

When computing with small examples, it may be handy to provide the elements of the Gröbner basis with a way of expressing them as elements in I, that is, as combinations of elements of the input. This can be done, not only for elements of G, but for any element, by the functions in the file trace.g. This file calls the file nparith.g for arithmetic keeping track of the expressions of polynomials as

combinations of elements from the original basis. With respect to a given input basis `B`, a polynomial `p` in the traced version is a record, called the traced polynomial, with two fields. One field, denoted `p.pol`, is the usual polynomial in NP format. The other, denoted `p.trace`, is a list of elements indexed by `B`. Each element of `p.trace` is a list whose elements are four-tuples `[ml,i,mr,c]` where `ml` and `mr` are monomials, `i` is an index of an element of `B` and `c` is a scalar. The interpretation of this data structure is that `p.pol` can be written as the sum over all four-tuples `[ml,i,mr,c]` of $c * ml * B_i * mr$. Functions for printing these expressions in a human understandable way are described in Section 3.7.

2.6 Truncation variant

For computations with large and/or infinite examples, it may be convenient to truncate everything above a certain degree. In fact, we encountered various examples where the polynomials are (weighted) homogeneous and then it makes perfect sense to truncate the polynomials, that is, to disregard everything above a certain degree. For then the Grobner basis, if it exists, will be also be homogeneous and the part consisting of all of its polynomials of degree less than a given degree d is equal to the Grobner basis of the join of the original list of polynomials with all monomials of degree $d+1$. Here an NP polynomial in n variables is called homogeneous of degree d with respect to v, a vector with non-negative integers of length n, if, for each of its monomials $[t_1,...,t_k]$, the sum over all v_{t_i} is equal to d. The most classical choice for v is the all-one vector in which case one often speaks of homogeneous without mentioning the all-one vector. If two polynomials are homogeneous with respect to v, then so are their S-polynomials. If K is a list of homogeneous polynomials with respect to v, then the normal form with respect to K of any homogeneous polynomial of degree d with respect to v is again homogeneous of degree d with respect to v. In particular, the Gröbner basis of a list of polynomials that are homogeneous with respect to v, consists of homogeneous polynomials, and those input polynomials contributing to polynomials in the Gröbner basis of degree at most d have degree at most d themselves. These facts enable the computation of the truncated Gröbner basis. The functions of this variant can be found in Section 3.8.

2.7 Module variant

Suppose we are given a finite set G of polynomials in a free non-commutative algebra A generated by, say t indeterminates, and a positive integer s. Denote by I the two-sided ideal of A generated by G. We can work with the free right A/I module $(A/I)^s$. See Section 2.2 on how to represent vectors of $(A/I)^s$ by elements of the free module A^s. Given a subset W of A^s, whose elements are called prefix relations, let W' be the submodule generated by the image of W in $(A/I)^s$. The function `SGrobnerModule` (3.9.1) is meant to determine the quotient module $(A/I)^s/W'$. If the algorithm terminates, it delivers a Gröbner basis for I as well as a suitable set of generators for W', with Gröbner like properties. This implies that `StrongNormalFormNPM` (3.9.5), a strong normal form computation, can be used to find the canonical representative in A^s of an element in $(A/I)^s/W'$. Theoretic details can be found in [Coh07]. If $(A/I)^s/W'$ is a finite-dimensional vector space over the coefficient field of A, then a basis can be found by use of `BaseQM` (3.9.2) and its dimension can be computed by use of `DimQM` (3.9.3).

2.8 Gröbner basis records

The function SGrobnerModule (3.9.1) calculates a Gröbner basis consisting of some two-sided relations in the algebra and some prefix or module relations in the vector space. These are returned in a record GBR. The two-sided relations can be found under the name GBR.ts and the prefix relations under the name GBR.p. Some other information is stored in this record as well.

The prefix conditions are in NPM format (see 2.2) and the two-sided relations are in NP format.

2.9 Quotient algebras

Once a Gröbner basis of a list G of polynomials in NP format, defining elements of a free algebra A, is computed, the quotient algebra QA of A by the two-sided ideal generated by G (or, which amounts to the same, the Gröbner basis) can be analyzed. A number of functions are available to determine whether QA is finite dimensional or not.

Elements of QA are represented by elements of A. Two elements are equal if and only if their strong normal forms coincide; see StrongNormalFormNP (3.5.6). The multiplication is take care of by MulQA (3.5.5), which is little more than the strong normal form of the product of two polynomials in NP format representing elements of QA.

If QA is finite dimensional, a basis of it over the field can be found by BaseQA (3.5.1). The size of the base, in other words, the dimension of QA, can be computed with DimQA (3.5.2). Right multiplication by an element of QA is a linear transformation. The matrix of this linear transformation with respect to the base, in case the element belongs to the base, can be computed by MatrixQA (3.5.3) or, for all basis elements, MatricesQA (3.5.4).

A list of leading terms of the Gröbner basis G can be constructed with LMonsNP (3.3.10). The dimension of QA only depends on this list and is computationally easier to work with than G. Most functions designed to analyze dimensionality work with a monomial ideal generated by a strong Gröbner basis, which in this case means that no element divides any other element.

The function FinCheckQA (3.6.2) determines whether QA is finite or infinite dimensional. More generally, the growth of QA can be determined by means of the function DetermineGrowthQA (3.6.1), which either returns the information that QA is finite dimensional, or that QA has polynomial growth, in which case it gives bounds for the degree of polynomial growth, or that QA has exponential growth. Finally, with the function HilbertSeriesQA (3.6.3) one can compute coefficients of the Hilbert series.

The purpose of the functions FinCheckQA (3.6.2) and DetermineGrowthQA (3.6.1) are closely related. The former is faster, while the latter provides more information, as illustrated from the following table.

	FinCheckQA	DetermineGrowthQA
finite	true	0
polynomial growth	false	d or [d1,d2]
exponential growth	false	"exponential growth"

Table: dimensionality functions; d stands for degree, [d1,d2] for an interval containing the degree

The function DetermineGrowthQA (3.6.1) may find the exact degree of polynomial growth (if at hand). If this is the case, that degree is returned. It may also happen that only an interval [d1,d2] is returned in which the dimension lies. To force an exact answer, its third argument should be true.

With the function PreprocessAnalysisQA (3.6.4), the computations done by these 3 functions can be sped up. Note however that by applying preprocessing of the data, the set of monomials in

the ideal basis is changed and corresponds no longer to the same quotient algebra (but to a quotient algebra with the same growth).

Chapter 3

Functions

3.1 Converting polynomials into different formats

3.1.1 GP2NP

▷ GP2NP(*gp*) (function)

Returns: If *gp* is an element of a free algebra, then the polynomial in NP format (see Section 2.1) corresponding to *gp*; if *gp* is an element of a free module, then the vector in NPM format (see Section 2.2) corresponding to *gp*.

This function will convert an element of a free algebra to a polynomial in NP format and an element of a free right module to a vector in NPM format.

Example: Let A be the free associative algebra with one over the rationals on the generators a and b. Let e be the one of the algebra.

```
gap> A:=FreeAssociativeAlgebraWithOne(Rationals,"a","b");;
gap> a:=A.a;;
gap> b:=A.b;;
gap> e:=One(A);;
gap> z:=Zero(A);;
```

Now let gp be the polynomial $ba - ab - e$.

```
gap> gp:=b*a-a*b-e;
(-1)*<identity ...>+(-1)*a*b+(1)*b*a
```

The polynomial in NP format, corresponding to gp can now be obtained with GP2NP:

```
gap> GP2NP(gp);
[ [ [ 2, 1 ], [ 1, 2 ], [  ] ], [ 1, -1, -1 ] ]
```

Let D be the free associative algebra over A of rank 2.

```
gap> D := A^2;;
```

Take the following list R of two elements of D.

```
gap> R := [ [b-e, z], [e+a*(e+a+b), -e-a*(e+a+b)] ];;
```

Convert the list R to a list of vectors in NPM format.

```
gap> List(R,GP2NP);
[ [ [ [ -1, 2 ], [ -1 ] ], [ 1, -1 ] ],
  [ [ [ -1, 1, 2 ], [ -1, 1, 1 ], [ -2, 1, 2 ], [ -2, 1, 1 ], [ -1, 1 ],
          [ -2, 1 ], [ -1 ], [ -2 ] ], [ 1, 1, -1, -1, 1, -1, 1, -1 ] ] ]
```

3.1.2 GP2NPList

▷ GP2NPList(*Lgp*) (function)

Returns: The list of polynomials in NP or NPM format corresponding to elements of a free algebra or module occurring in the list *Lgp*.

This function has the same effect as List(Lgp,GBNP).

Example: Let A be the free associative algebra with one over the rationals on the generators a and b. Let e be the one of the algebra.

```
gap> A:=FreeAssociativeAlgebraWithOne(Rationals,"a","b");;
gap> a:=A.a;;
gap> b:=A.b;;
gap> e:=One(A);;
```

Let Lgp be the list of polynomials $[a^2-e, b^2-e, ba-ab-e]$.

```
gap> Lgp:=[a^2-e,b^2-e,b*a-a*b-e];
[ (-1)*<identity ...>+(1)*a^2, (-1)*<identity ...>+(1)*b^2,
  (-1)*<identity ...>+(-1)*a*b+(1)*b*a ]
```

The polynomial in NP format corresponding to gp can be obtained with GP2NP:

```
gap> GP2NPList(Lgp);
[ [ [ [ 1, 1 ], [  ] ], [ 1, -1 ] ], [ [ [ 2, 2 ], [  ] ], [ 1, -1 ] ],
  [ [ [ 2, 1 ], [ 1, 2 ], [  ] ], [ 1, -1, -1 ] ] ]
```

The same result is obtained by a simple application of the standard List function in GAP:

```
gap> List(Lgp,GP2NP) = GP2NPList(Lgp);
true
```

3.1.3 NP2GP

▷ NP2GP(*np*, *A*) (function)

Returns: The GAP format of the polynomial *np* in NP format.

This function will convert a polynomial in NP format to a GAP polynomial in the free associative algebra *A* and a vector in NPM format to a GAP vector in the free module *A*. In case of the NP format, the number of variables should not exceed the rank of the free algebra *A*. In case of the NPM format, the absolute of the negative numbers should not exceed the rank of the free module *A*.

Example: Let A be the free associative algebra with one over the rationals on the generators a and b.

```
gap> A:=FreeAssociativeAlgebraWithOne(GF(3),"a","b");;
```

Let np be a polynomial in NP format.

```
gap> np:=[ [ [ 2, 1 ], [ 1, 2 ], [  ] ], [ Z(3)^0, Z(3), Z(3) ] ];;
```

The polynomial can be converted to the corresponding element of *A* with NP2GP:

```
gap> NP2GP(np,A);
(Z(3)^0)*b*a+(Z(3))*a*b+(Z(3))*<identity ...>
```

Note that some information of the coefficient field of a polynomial np in NP format can be obtained from the second list of np.

```
gap> One(np[2][1]);
Z(3)^0
```

Now let M be the module A^2 and let npm be a polynomial over that module in NPM form.

```
gap> M:=A^2;;
gap> npm:=[ [ [ -1, 1 ], [ -2, 2 ] ], [ Z(3)^0, Z(3)^0 ] ];;
```

The element of *M* corresponding to npm is

```
gap> NP2GP(npm,M);
[ (Z(3)^0)*a, (Z(3)^0)*b ]
```

If M is a module of dimension 2 over A and Lnp a list of polynomials in NPM format, then the polynomials can be converted to the corresponding polynomials of M as follows:

```
gap> M:=A^2;;
gap> Lnp:=[ [ [ [ -2, 1, 1 ], [ -2, 1 ] ], [ 1, -1 ] ],
>   [ [ [ -1, 2, 2], [-2, 1 ] ], [ 1, -1 ]*Z(3)^0 ] ];;
gap> List(Lnp, m -> NP2GP(m,M));
[ [ <zero> of ..., (Z(3))*a+(Z(3)^0)*a^2 ], [ (Z(3)^0)*b^2, (Z(3))*a ] ]
```

3.1.4 NP2GPList

▷ NP2GPList(*Lnp*, *A*) (function)

Returns: The list of polynomials corresponding to *Lnp* in GAP format.

This function will convert the list *Lnp* of polynomials in NP format to a list of GAP polynomials in the free associative algebra *A*.

Example: Let A be the free associative algebra with one over the rationals on the generators a and b.

```
gap> A:=FreeAssociativeAlgebraWithOne(Rationals,"a","b");;
```

Let Lnp be a list of polynomials in NP format. Then Lnp can be converted to a list of polynomials of A with NP2GPList:

```
gap> Lnp:=[ [ [ [ 1, 1, 1 ], [ 1 ] ], [ 1, -1 ] ],
>   [ [ [ 2, 2 ], [ ] ], [ 1, -1 ] ] ];;
gap> NP2GPList(Lnp,A);
[ (1)*a^3+(-1)*a, (1)*b^2+(-1)*<identity ...> ]
```

It has the same effect as the function `List` applied as follows.

```
gap> List(Lnp, p -> NP2GP(p,A));
[ (1)*a^3+(-1)*a, (1)*b^2+(-1)*<identity ...> ]
```

Now let `M` be a module of dimension 2 over `A` and `Lnp` a list of vectors in NPM format. Then polynomials `Lnp` can be converted to the corresponding vectors of `M` with NP2GPList:

```
gap> M:=A^2;;
gap> Lnp:=[ [ [ [ -2, 1, 1 ], [ -2, 1 ] ], [ 1, -1 ] ],
>   [ [ [ -1, 1 ], [ -2 ] ], [ 1, -1 ] ] ];;
gap> NP2GPList(Lnp,M);
[ [ <zero> of ..., (-1)*a+(1)*a^2 ], [ (1)*a, (-1)*<identity ...> ] ]
```

The same result can be obtained by application of the standard List function:

```
gap> List(Lnp, m -> NP2GP(m,M)) = NP2GPList(Lnp,M) ;
true
```

3.2 Printing polynomials in NP format

3.2.1 PrintNP

▷ `PrintNP(`*np*`)` (function)

This function prints a polynomial *np* in NP format, using the letters `a`, `b`, `c`, ... for x_1, x_2, x_3, ..., except that everything beyond l (the 12-th letter) is printed as x.

This function prints a polynomial *np* in NP format as configured by the function `GBNP.ConfigPrint` (3.2.2).

Example: Consider the following polynomial in NP format.

```
gap> p := [[[1,1,2],[1,2,2],[]],[1,-2,3]];;
```

It can be printed in the guise of a polynomial in `a` and `b` by the function `PrintNP`:

```
gap> PrintNP(p);
 a^2b - 2ab^2 + 3
```

3.2.2 GBNP.ConfigPrint

▷ `GBNP.ConfigPrint(arg)` (function)

By default the generators of the algebra are printed as `a`, ..., `l` and everything after the twelfth generator as `x`. By calling `ConfigPrint` it is possible to alter this printing convention. The argument(s) will be an algebra or arguments used for naming algebras in GAP upon creation. More specifically, we have the following choices.

no arguments
When the function is invoked without arguments the printing is reset to the default (see above).

algebra
When the function is invoked with an algebra as argument, generators will be printed as they would be in the algebra.

algebra,integer
When the function is invoked with an algebra and an integer *n* as arguments, generators will be printed as they would be in the algebra and separated over the *n* dimensions.

leftmodule
When the function is invoked with an leftmodule A^n of an associative algebra as argument, generators will be printed as they would be in the algebra, separated over the *n* dimensions.

string
When the function is invoked with a string as its argument, it is assumed that there is only 1 generator and that this should be named as indicated by the string.

integer
When the function is invoked with an integer as its argument, the *n*-th generator will be printed as `x.<n>`.

integer, string
When the function is invoked with a non-negative integer and a string as its arguments, generators will be printed as `<s>.<n>`, where `<s>` is the string given as argument and `<n>` the number of the generator. There is no checking whether the number given as argument is really the dimension. So it is possible that higher numbers return in the output. This way of input is useful however, because it is a distinction from the one-dimensional case and compatible with the way a free algebra is created.

string, string, ..., string
When the function is invoked with a sequence of strings, then generators will be printed with the corresponding string or `x` if the sequence is not long enough.

Example: Consider the following two polynomials in NP format.

```
gap> p1 := [[[1,1,2],[]],[1,-1]];;
gap> p2 := [[[1,2,2],[]],[1,-1]];;
```

They can be printed by the function `PrintNP`.

```
gap> PrintNP(p1);
 a^2b - 1
gap> PrintNP(p2);
 ab^2 - 1
```

We can let the variables be printed as `x` and `y` instead of `a` and `b` by means of `GBNP.ConfigPrint`.

```
gap> GBNP.ConfigPrint("x","y");
gap> PrintNP(p1);
 x^2y - 1
gap> PrintNP(p2);
 xy^2 - 1
```

We can also let the variables be printed as `x.1` and `x.2` instead of `a` and `b` by means of `GBNP.ConfigPrint`.

```
gap> GBNP.ConfigPrint(2,"x");
gap> PrintNP(p1);
 x.1^2x.2 - 1
gap> PrintNP(p2);
 x.1x.2^2 - 1
```

We can even assign strings to the variables to be printed like `alice` and `bob` instead of `a` and `b` by means of `GBNP.ConfigPrint`.

```
gap> GBNP.ConfigPrint("alice","bob");
gap> PrintNP(p1);
 alice^2bob - 1
gap> PrintNP(p2);
 alicebob^2 - 1
```

Alternatively, we can introduce the free algebra `A` with two generators, and print the polynomials as members of `A`:

```
gap> A:=FreeAssociativeAlgebraWithOne(Rationals,"a","b");;
gap> GBNP.ConfigPrint(A);
gap> PrintNP(p1);
 a^2b - 1
gap> PrintNP(p2);
 ab^2 - 1
```

3.2.3 PrintNPList

▷ `PrintNPList(`*Lnp*`)` (function)

This function prints a list *Lnp* of polynomials in NP format, using the function `PrintNP`.
Example: We put two polynomials in NP format into the list `Lnp`.

```
gap> p1 := [[[1,1,2],[]],[1,-1]];;
gap> p2 := [[[1,2,2],[]],[1,-1]];;
gap> Lnp := [p1,p2];;
```

We can print the list with `PrintNPList`.

```
gap> PrintNPList(Lnp);
 a^2b - 1
 ab^2 - 1
```

Alternatively, using the function `GBNP.ConfigPrint` (3.2.2), we can introduce the free algebra *A* with two generators, and print the polynomials of the list as members of *A*:

```
gap> A:=FreeAssociativeAlgebraWithOne(Rationals,"a","b");;
gap> GBNP.ConfigPrint(A);
gap> PrintNPList(Lnp);
 a^2b - 1
 ab^2 - 1
```

3.3 Calculating with polynomials in NP format

3.3.1 NumAlgGensNP

▷ `NumAlgGensNP(`*np*`)` (function)

Returns: The minimum number `t` so that *np* belongs to the free algebra on `t` generators.

When called with an NP polynomial *np*, this function returns the minimum number of generators needed for the corresponding algebra to contain the *np*. If *np* is a polynomial without generators, that is, equivalent to 0 or 1, then `0` is returned.

Example: Consider the following polynomial in NP format.

```
gap> np := [[[2,2,2,1,1,1],[4],[3,2,3]],[1,-3,2]];;
gap> PrintNP(np);
 b^3a^3 - 3d + 2cbc
gap> NumAlgGensNP(np);
4
```

3.3.2 NumAlgGensNPList

▷ `NumAlgGensNPList(`*Lnp*`)` (function)

Returns: The minimum number `t` so that each polynomial in *Lnp* belongs to the free algebra on `t` generators.

When called with a list of NP polynomials *Lnp*, this function returns the minimum number of generators needed for the corresponding algebra to contain the NP polynomials in *Lnp*. If *Lnp* only contains polynomials without generators, that is equivalent to 0 and 1, then `0` is returned.

Example: Consider the following two polynomials in NP format.

```
gap> p1 := [[[1,1,2,3,1],[2],[1]],[1,-2,1]];;
gap> p2 := [[[2,2,1,4,3],[]],[1,-1]];;
gap> PrintNPList([p1,p2]);
 a^2bca - 2b + a
 b^2adc - 1
gap> NumAlgGensNPList([p1,p2]);
4
```

3.3.3 NumModGensNP

▷ NumModGensNP(*npm*) (function)

Returns: The minimum number mt so that *npm* belongs to the free module on mt generators.

When called with a polynomial *npm* in NPM format, this function returns the minimum number of module generators needed for the corresponding algebra to contain *npm*. If *npm* is an NP polynomial that does not contain module generators, then 0 is returned.

Example: Consider the following polynomial in NPM format.

```
gap> np := [[[-1,1,2,3,1],[-2],[-1]],[1,-2,1]];;
gap> PrintNP(np);
[ abca + 1 , - 2 ]
gap> NumModGensNP(np);
2
```

3.3.4 NumModGensNPList

▷ NumModGensNPList(*Lnpm*) (function)

Returns: The minimum number mt so that each member of *npm* belongs to the free module on mt generators.

When called with a list of polynomials *Lnpm* in NPM format, this function returns the minimum number of module generators needed to contain the polynomials in *Lnpm*. If there are only polynomials in *Lnpm* in NP format, then 0 is returned.

Example: Consider the following two polynomials in NPM format.

```
gap> v1 := [[[-1,1,2,3,1],[-2],[-1]],[1,-2,1]];;
gap> v2 := [[[-2,2,1,4,3],[-3]],[1,-1]];;
gap> PrintNPList([v1,v2]);
[ abca + 1 , - 2 ]
[ 0, badc , - 1 ]
gap> NumModGensNPList([v1,v2]);
3
```

3.3.5 AddNP

▷ AddNP(*u*, *v*, *c*, *d*) (function)

Returns: *c*∗*u*+*d*∗*v*

Computes *c*∗*u*+*d*∗*v* where *u* and *v* are polynomials in NP format and *c* and *d* are scalars.

Example: Consider the following two polynomials in NP format.

```
gap> p1 := [[[1,1,2],[]],[1,-3]];;
gap> p2 := [[[1,2,2],[]],[1,-4]];;
```

The second can be subtracted from the first by the function AddNP.

```
gap> PrintNP(AddNP(p1,p2,1,-1));
 - ab^2 + a^2b + 1
```

3.3.6 BimulNP

▷ BimulNP(*ga, np, dr*) (function)

Returns: the polynomial *ga*∗*np*∗*dr* in NP format

When called with a polynomial *np* and two monomials *ga*, *dr*, the function will return *ga*∗*np*∗*dr*. Recall from Section 2.1 that monomials are represented as lists.

Example: Consider the following two polynomials in NP format.

```
gap> p1 := [[[1,1,2],[]],[1,-3]];;
gap> p2 := [[[1,2,2],[]],[1,-4]];;
```

Multiplying p1 from the right by b and multiplying p2 from the left by a is possible with the function BimulNP:

```
gap> PrintNP(BimulNP([],p1,[2]));
 a^2b^2 - 3b
gap> PrintNP(BimulNP([1],p2,[]));
 a^2b^2 - 4a
```

3.3.7 CleanNP

▷ CleanNP(*u*) (function)

Returns: The cleaned up version of *u*

Given a polynomial in NP format, this function collects terms with same monomial, removes trivial monomials, and orders the monomials, with biggest one first.

Example: Consider the following polynomial in NP format.

```
gap> p := [[[1,1,2],[],[1,1,2],[]],[1,-3,-2,3]];;
gap> PrintNP(p);
 a^2b - 3 - 2a^2b + 3
```

The monomials [1,1,2] and [] occur twice each. For many functions this representation of a polynomial in NP format is not allowed. It needs to be cleaned, as by CleanNP:

```
gap> PrintNP(CleanNP(p));
 - a^2b
```

In order to define a polynomial over $GF(2)$, the coefficients need to be defined over this field. Such a list of coefficients can be obtained in GAP from a list of integers by multiplying with the identity element of the field. The resulting polynomial need not be clean, and so should be made clean again with CleanNP.

```
gap> p := [[[1,1,2],[]],One(GF(2))*[1,-2]];;
gap> CleanNP(p);
[ [ [ 1, 1, 2 ] ], [ Z(2)^0 ] ]
```

3.3.8 GtNP

▷ `GtNP(u, v)` (function)

Returns: `true` if $u > v$ and `false` if $u \leq v$

Greater than function for monomials u and v represented as in Section 2.1. It tests whether $u > v$. The ordering is done by degree and then lexicographically.

Example: Consider the following two monomials.

```
gap> u := [1,1,2];
[ 1, 1, 2 ]
gap> v := [2,2,1];
[ 2, 2, 1 ]
```

We test whether u is greater than v.

```
gap> GtNP(u,v);
false
```

3.3.9 LtNP

▷ `LtNP(u, v)` (function)

Returns: `true` if $u < v$ and `false` if $u \geq v$

Less than function for NP monomials, tests whether $u < v$. The ordering is done by degree and then lexicographically.

Example: Consider the following two monomials.

```
gap> u := [1,1,2];
[ 1, 1, 2 ]
gap> v := [2,2,1];
[ 2, 2, 1 ]
```

We test whether u is less than v.

```
gap> LtNP(u,v);
true
```

3.3.10 LMonNP

▷ `LMonNP(Lnp)` (function)

▷ `LMonsNP(Lnp)` (function)

Returns: The leading monomial or a list of leading monomials.

These functions return the leading monomial of a polynomial (resp. the leading monomials of a list of polynomials) in NP format. The polynomials of *Lnp* are required to be clean; see Section 3.3.7.

Example: We put two polynomials in NP format into the list Lnp.

```
gap> p1 := [[[1,1,2],[]],[1,-1]];;
gap> p2 := [[[1,2,2],[]],[1,-1]];;
gap> Lnp := [p1,p2];;
```

The list of leading monomials is computed by LMonsNP:

```
gap> LMonsNP(Lnp);
[ [ 1, 1, 2 ], [ 1, 2, 2 ] ]
```

For a nicer printing, the monomials can be converted into polynomials in NP format, and then submitted to PrintNPList:

```
gap> PrintNPList(List(LMonsNP(Lnp), q -> [[q],[1]]));
 a^2b
 ab^2
```

3.3.11 LTermNP

▷ LTermNP(*Lnp*) (function)

▷ LTermsNP(*Lnp*) (function)

Returns: The leading term or a list of leading terms.

These functions return the leading term of a polynomial (resp. the leading terms of a list of polynomials) in NP format. The polynomials of *Lnp* are required to be clean; see Section 3.3.7.

Example

```
gap> p1 := [[[1,1,2],[1]],[6,-7]];;
gap> p2 := [[[1,2,2],[2]],[8,-9]];;
gap> Lnp := [p1,p2];;
gap> LTermNP( p1 );
[ [ [ 1, 1, 2 ] ], [ 6 ] ]
gap> LTnp := LTermsNP( Lnp );
[ [ [ [ 1, 1, 2 ] ], [ 6 ] ], [ [ [ 1, 2, 2 ] ], [ 8 ] ] ]
gap> PrintNPList( LTnp );
6a^2b
8ab^2
```

3.3.12 MkMonicNP

▷ MkMonicNP(*np*) (function)

Returns: *np* made monic

This function returns the scalar multiple of a polynomial *np* in NP format that is monic, i.e., has leading coefficient equal to 1.

Example: Consider the following polynomial in NP format.

```
gap> p := [[[1,1,2],[]],[2,-1]];;
gap> PrintNP(p);
 2a^2b - 1
```

The coefficient of the leading term is 2. The function MkMonicNP finds this coefficient and divides every term by it:

```
gap> PrintNP(MkMonicNP(p));
 a^2b - 1/2
```

3.3.13 FactorOutGcdNP

▷ `FactorOutGcdNP(np)` (function)

Returns: *np* with Gcd(coefficients) factored out

This function returns the scalar multiple of a polynomial *np* in NP format with integer coefficients such that its coefficients have Gcd equal to 1. If the coefficients are not all integers then `fail` is returned.

Example: Consider the following polynomial in NP format.

```
gap> p := [[[1,1,2],[1,2],[1]],[30,70,105]];;
gap> PrintNP(p);
 30a^2b + 70ab + 105a
```

The `Gcd` of the coefficients $[30, 70, 105]$ is 5. The function `FactorOutGcdNP` divides the polynomial by 5:

```
gap> PrintNP(FactorOutGcdNP(p));
 6a^2b + 14ab + 21a
gap> m := MkMonicNP(p);
[ [ [ 1, 1, 2 ], [ 1, 2 ], [ 1 ] ], [ 1, 7/3, 7/2 ] ]
gap> fm := FactorOutGcdNP(m);
fail
```

3.3.14 MulNP

▷ `MulNP(np1, np2)` (function)

Returns: *np1*∗*np2*

When invoked with two polynomials *np1* and *np2* in NP format, this function will return the product *np1*∗*np2*.

Example: Consider the following two polynomials in NP format.

```
gap> p1 := [[[1,1,2],[]],[1,-1]];;
gap> p2 := [[[1,2,2],[]],[1,-1]];;
```

The function `MulNP` multiplies the two polynomials.

```
gap> PrintNP(MulNP(p1,p2));
 a^2bab^2 - ab^2 - a^2b + 1
```

The fact that this multiplication is not commutative is illustrated by the following comparison, using `MulNP` twice and `AddNP` once.

```
gap> PrintNP(AddNP(MulNP(p1,p2),MulNP(p2,p1),1,-1));
 - ab^2a^2b + a^2bab^2
```

3.4 Gröbner functions, standard variant

3.4.1 Grobner

▷ `Grobner(`*`Lnp[, D][, max]`*`)` (function)

Returns: If the algorithm terminates, a Gröbner Basis or a record if *max* is specified (see description).

For a list *Lnp* of polynomials in NP format this function will use Buchberger's algorithm with normal form to find a Gröbner Basis (if possible, the general problem is unsolvable).

When called with the optional argument *max*, which should be a positive integer, the calculation will be interrupted if it has not ended after *max* iterations. The return value will be a record containing lists `G` and `todo` of polynomials in NP format, a boolean `completed`, and an integer `iterations`. Here `G` and `todo` form a Gröbner pair (see [Coh07]). The number of performed iterations will be placed in `iterations`. If the algorithm has terminated, then `todo` will be the empty list and `completed` will be equal to `true`. If the algorithm has not terminated, then `todo` will be a non-empty list of polynomials in NP format and `completed` will be `false`.

By use of the optional argument *D*, it is possible to resume a previously interrupted calculation.

Example: Consider the following two polynomials in NP format.

```
gap> p1 := [[[1,1,2],[]],[1,-1]];;
gap> p2 := [[[1,2,2],[]],[1,-1]];;
gap> PrintNPList([p1,p2]);
 a^2b - 1
 ab^2 - 1
```

Their Gröbner basis can be computed by the function `Grobner`.

```
gap> G := Grobner([p1,p2]);;
gap> PrintNPList(G);
 b - a
 a^3 - 1
```

One iteration of the Gröbner computations is invoked by use of the parameter `max`:

```
gap> R := Grobner([p1,p2],1);;
gap> PrintNPList(R.G);
 b - a
gap> PrintNPList(R.todo);
 a^3 - 1
gap> R.iterations;
1
gap> R.completed;
false
```

The above list `R.todo` can be used to resume the computation of the Gröbner basis computation with the Gröbner pair `R.G`, `R.todo`:

```
gap> PrintNPList(Grobner(R.G,R.todo));
 b - a
 a^3 - 1
```

In order to perform the Gröbner basis computation with polynomials in a free algebra over the field $GF(2)$, the coefficients of the polynomials need to be defined over that field.

```
gap> PrintNPList(Grobner([[p1[1],One(GF(2))*p1[2]],[p2[1],One(GF(2))*p1[2]]]));
 b + a
 a^3 + Z(2)^0
```

3.4.2 SGrobner

▷ SGrobner(*Lnp[, todo][, max]*) (function)

Returns: If the algorithm terminates, a Gröbner Basis or a record if *max* is specified (see description).

For a list *Lnp* of polynomials in NP format this function will use Buchberger's algorithm with strong normal form (see [Coh07]) to find a Gröbner Basis (if possible, the general problem is unsolvable).

When called with the optional argument *max*, which should be a positive integer, the calculation will be interrupted if it has not ended after *max* iterations. The return value will be a record containing lists `G` and `todo` of polynomials in NP format, a boolean `completed`, and an integer `iterations`. Here `G` and `todo` form a Gröbner pair (see [Coh07]). The number of performed iterations will be placed in `iterations`. If the algorithm has terminated, then `todo` will be the empty list and `completed` will be equal to `true`. If the algorithm has not terminated, then `todo` will be a non-empty list of polynomials in NP format and `completed` will be `false`.

By use of the optional argument *D*, it is possible to resume a previously interrupted calculation.

Example: Consider the following two polynomials in NP format.

```
gap> p1 := [[[1,1,2],[]],[1,-1]];;
gap> p2 := [[[1,2,2],[]],[1,-1]];;
gap> PrintNPList([p1,p2]);
 a^2b - 1
 ab^2 - 1
```

Their Gröbner basis can be computed by the function `Grobner`.

```
gap> G := SGrobner([p1,p2]);;
gap> PrintNPList(G);
 b - a
 a^3 - 1
```

One iteration of the Gröbner computations is invoked by use of the parameter `max`:

```
gap> R := SGrobner([p1,p2],1);;
gap> PrintNPList(R.G);
 b - a
gap> PrintNPList(R.todo);
 a^3 - 1
gap> R.iterations;
1
gap> R.completed;
false
```

The above list R.todo can be used to resume the computation of the Gröbner basis computation with the Gröbner pair R.G, R.todo:

```
gap> PrintNPList(SGrobner(R.G,R.todo));
 b - a
 a^3 - 1
```

3.4.3 IsGrobnerBasis

▷ IsGrobnerBasis(*G*) (function)

Returns: true if *G* is a Gröbner basis and false otherwise

When invoked with a list *G* of polynomials in NP format (see Section 2.1), this function will check whether the list is a Gröbner basis. The check is based on Theorem 1.4 from [Coh07].

Polynomials representing zero are allowed in *G*.

Example: Consider the following list of two polynomials in NP format.

```
gap> Lnp := [[[1,1,2],[]],[1,-1]], [[[1,2,2],[]],[1,-1]]];;
gap> PrintNPList(Lnp);
 a^2b - 1
 ab^2 - 1
```

The function IsGrobner checks whether the list is a Gröbner basis.

```
gap> IsGrobnerBasis(Lnp);
false
```

So the answer should be true for the result of a Gröbner computation:

```
gap> IsGrobnerBasis(Grobner(Lnp));
true
```

3.4.4 IsStrongGrobnerBasis

▷ IsStrongGrobnerBasis(*G*) (function)

Returns: true if *G* is a strong Gröbner basis and false otherwise

When invoked with a list *G* of polynomials in NP format (see Section 2.1), this function will check whether the polynomials in this list form a strong Gröbner basis (see [Coh07]).

Polynomials representing zero are allowed in *G*.

Example: Consider the following list of two polynomials in NP format.

```
gap> Lnp := [[[1,1,2],[]],[1,-1]], [[[1,2,2],[]],[1,-1]]];;
gap> PrintNPList(Lnp);
 a^2b - 1
 ab^2 - 1
```

The function IsStrongGrobner checks whether the list is a strong Gröbner basis.

```
gap> IsStrongGrobnerBasis(Lnp);
false
```

But the answer should be `true` for the result of a strong Gröbner computation:

```
gap> IsStrongGrobnerBasis(SGrobner(Lnp));
true
```

A Gröbner basis that is not a strong Gröbner basis:

```
gap> B := SGrobner(Lnp);;
gap> Add(B,AddNP(Lnp[1],B[1],1,-1));;
gap> PrintNPList(B);
 b - a
 a^3 - 1
 a^2b - b + a - 1
gap> IsGrobnerBasis(B);
true
gap> IsStrongGrobnerBasis(B);
false
```

3.4.5 IsGrobnerPair

▷ `IsGrobnerPair(G, D)` (function)

Returns: A boolean, which has the value `true` if the input forms a Gröbner pair

When called with two lists of polynomials in NP format, this function returns true if they form a Gröbner pair. Testing whether `D` is a basic set for `G` might involve computing the Gröbner basis. Instead of this only some simple computations are done to see if it can easily be proven that `D` is a basic set for `G`. If this cannot be proven easily, then `false` is returned, even though G, D might still be a Gröbner pair.

Example: Consider the following four polynomials in NP format.

```
gap> p1 := [[[1,1,2],[]],[1,-1]];;
gap> p2 := [[[1,2,2],[]],[1,-1]];;
gap> q1 := [[[2],[1]],[1,-1]];;
gap> q2 := [[[1,1,1],[]],[1,-1]];;
```

The function `IsGrobnerPair` is used to check whether some combinations of these polynomials in two lists provide Gröbner pairs.

```
gap> IsGrobnerPair([p1,p2,q1],[q2]);
true
gap> IsGrobnerPair([q1,q2],[p1,p2]);
false
```

The function `IsGrobnerPair` applied with an empty list as second argument is a check whether the first argument is a Gröbner basis.

```
gap> IsGrobnerPair([p1,p2],[]) = IsGrobnerBasis([p1,p2]);
true
```

3.4.6 MakeGrobnerPair

▷ `MakeGrobnerPair(G, D)` (function)

Returns: A record containing a new Grobner pair

When called with as arguments a pair G, D, this function cleans `G` and `D` and adds some obstructions to `D` till it is easily provable that `D` is a basic set for `G` (see [Coh07]). The result is a record containing the fields `G` and `todo` representing the Gröbner pair.

Example: Consider the following two polynomials in NP format.

```
gap> p1 := [[[1,1,2],[]],[1,-1]];;
gap> p2 := [[[1,2,2],[]],[1,-1]];;
```

The function `MakeGrobnerPair` turns the list with these two polynomials into a Gröbner pair, once the empty list is added as a second argument. The result is a record whose fields `G` and `todo`

```
gap> GP := MakeGrobnerPair([p1,p2],[]);;
gap> PrintNPList(GP.G);
 a^2b - 1
 ab^2 - 1
gap> PrintNPList(GP.todo);
 b - a
```

These fields are ready for use in `Grobner`

```
gap> GB := Grobner(GP.G,GP.todo);;
gap> PrintNPList(GB);
 b - a
 a^3 - 1
```

3.5 Finite-dimensional quotient algebras

3.5.1 BaseQA

▷ `BaseQA(G, t, maxno)` (function)

Returns: A list of terms forming a basis of the quotient algebra of the (non-commutative) polynomial algebra in `t` variables by the 2-sided ideal generated by `G`

When called with a Gröbner basis `G`, the number `t` of generators of the algebra, and a maximum number of terms to be found `maxno`, BaseQA will return a (partial) base of the quotient algebra. If this function is invoked with `maxno` equal to 0, then a full basis will be given. If the dimension of this quotient algebra is infinite and `maxno` is set to 0, then the algorithm behind this function will not terminate.

Example: Consider the following Gröbner basis.

```
gap> p1 := [[[1,1,2],[]],[1,-1]];;
gap> p2 := [[[1,2,2],[]],[1,-1]];;
gap> G := Grobner([p1,p2]);;
gap> PrintNPList(G);
 b - a
 a^3 - 1
```

The function BaseQA computes a basis for the quotient algebra of the free algebra over the rationals with generators a and b by the two-sided ideal generated by G.

```
gap> PrintNPList(G);
 b - a
 a^3 - 1
gap> BaseQA(G,2,0);
[ [ [ [  ] ], [ 1 ] ], [ [ [ 1 ] ], [ 1 ] ], [ [ [ 1, 1 ] ], [ 1 ] ] ]
gap> PrintNPList(BaseQA(G,2,0));
 1
 a
 a^2
```

It is necessary for a correct result that the first argument be a Gröbner basis, as will be clear from the following invocation of BaseQA.

```
gap> PrintNPList(BaseQA([p1,p2],2,10));
 1
 a
 b
 a^2
 ab
 ba
 b^2
 a^3
 aba
 ba^2
 bab
 b^2a
 b^3
```

3.5.2 DimQA

▷ DimQA(*G*, *t*) (function)

Returns: The dimension of the quotient algebra

When called with a Gröbner basis *G* and a number of variables *t*, the function DimQA will return the dimension of the quotient algebra of the free algebra generated by *t* variables by the ideal generated by *G* if it is finite. It will not terminate if the dimension is infinite.

If *t*=0, the function will compute the minimal value of t such that the polynomials in *G* belong to the free algebra on t generators.

To check whether the dimension of the quotient algebra is finite and to determine the type of growth if it is infinite, see also the functions FinCheckQA (3.6.2) and DetermineGrowthQA (3.6.1) in Section 3.6.

Example: Consider the following Gröbner basis.

```
gap> p1 := [[[1,1,2],[]],[1,-2]];;
gap> p2 := [[[1,2,2],[]],[1,-2]];;
gap> G := Grobner([p1,p2]);;
gap> PrintNPList(G);
 b - a
 a^3 - 2
```

The function DimQA computes the dimension of the quotient algebra of the free algebra over the rationals with generators a and b by the two-sided ideal generated by G.

```
gap> DimQA(G,2);
3
```

3.5.3 MatrixQA

▷ MatrixQA(*i*, *B*, *GB*) (function)

Returns: The matrix representation for the *i*-th generator of the algebra for right multiplication in the quotient algebra

Given a basis *B* of the quotient algebra, a Gröbner basis (record) *GB*, and a natural number *i*, this function creates a matrix representation for the *i*-th generator of the algebra for right multiplication.

Example: Consider the following two polynomials in NP format.

```
gap> p1 := [[[1,1,1,2],[]],[1,-1]];;
gap> p2 := [[[2,2,2,1],[]],[1,-1]];;
```

The matrix of right multiplication by the first indeterminate a on the quotient algebra with respect to the ideal generated by p1 and p2 is obtained by applying MatrixQA to the Gröbner basis of these generators and a basis of the quotient algebra as found in BaseQA (3.5.1):

```
gap> GB := Grobner([p1,p2]);;
gap> B := BaseQA(GB,2,0);;
gap> PrintNPList(B);
 1
 a
 b
 a^2
 ab
 a^3
 a^2b
 a^4
gap> Display(MatrixQA(1, B,GB));
[ [  0,  1,  0,  0,  0,  0,  0,  0 ],
  [  0,  0,  0,  1,  0,  0,  0,  0 ],
  [  0,  0,  0,  0,  1,  0,  0,  0 ],
  [  0,  0,  0,  0,  0,  1,  0,  0 ],
  [  0,  0,  0,  0,  0,  0,  1,  0 ],
  [  0,  0,  0,  0,  0,  0,  0,  1 ],
  [  1,  0,  0,  0,  0,  0,  0,  0 ],
  [  0,  0,  1,  0,  0,  0,  0,  0 ] ]
```

The function is also applicable to Gröbner basis records for modules. Consider the following two vectors.

```
gap> v1 := [[[-1,1,2],[-1]],[1,-1]];;
gap> v2 := [[[-2,2,2],[-2]],[1,-2]];;
```

The Gröbner basis record for this data is found by SGrobnerModule (3.9.1) and a quotient module basis by BaseQM (3.9.2):

```
gap> GBR := SGrobnerModule([v1,v2],[p1,p2]);;
gap> B := BaseQM(GBR,2,2,0);;
```

The matrix of right multiplication by a, the first generator of the free algebra, is

```
gap> Display(MatrixQA(1,B,GBR));
[ [  0,  1 ],
  [  1,  0 ] ]
```

3.5.4 MatricesQA

▷ MatricesQA(*t, B, GB*) (function)

Returns: The matrix representation for the *t* generators of the algebra for right multiplication in the quotient algebra

Given a basis *B* of the quotient algebra, a Gröbner basis (record) *GB*, and a natural number *t*, this function creates a list of *t* matrices representing the linear transformations of the generators of the algebra by right multiplication on the quotient algebra.

Example: Consider the following two polynomials in NP format.

```
gap> p1 := [[[1,1,1,2],[]],[1,-1]];;
gap> p2 := [[[2,2,2,1],[]],[1,-1]];;
```

The function MatricesQA gives the list of matrices found by MatrixQA (3.5.3) when the first argument takes the integer values between 1 and the number of all algebra generators.

```
gap> GB := Grobner([p1,p2]);;
gap> B := BaseQA(GB,2,0);;
gap> mats := MatricesQA(2,B,GB);;
gap> for mat in mats do Display(mat); Print("\n"); od;
[ [  0,  1,  0,  0,  0,  0,  0,  0 ],
  [  0,  0,  0,  1,  0,  0,  0,  0 ],
  [  0,  0,  0,  0,  1,  0,  0,  0 ],
  [  0,  0,  0,  0,  0,  1,  0,  0 ],
  [  0,  0,  0,  0,  0,  0,  1,  0 ],
  [  0,  0,  0,  0,  0,  0,  0,  1 ],
  [  1,  0,  0,  0,  0,  0,  0,  0 ],
  [  0,  0,  1,  0,  0,  0,  0,  0 ] ]

[ [  0,  0,  1,  0,  0,  0,  0,  0 ],
  [  0,  0,  0,  0,  1,  0,  0,  0 ],
  [  0,  0,  0,  1,  0,  0,  0,  0 ],
  [  0,  0,  0,  0,  0,  0,  1,  0 ],
  [  0,  0,  0,  0,  0,  1,  0,  0 ],
  [  1,  0,  0,  0,  0,  0,  0,  0 ],
  [  0,  0,  0,  0,  0,  0,  0,  1 ],
  [  0,  1,  0,  0,  0,  0,  0,  0 ] ]
```

The result is also obtainable by use of the List function:

```
gap> MatricesQA(2,B,GB) = List([1,2], q -> MatrixQA(q,B,GB));
true
```

3.5.5 MulQA

▷ `MulQA(p1, p2, G)` (function)

Returns: The strong normal form of the product `p1*p2` with respect to `G`

When called with two polynomials in NP form, `p1` and `p2`, and a Gröbner basis `G`, this function will return the product in the quotient algebra.

Example: Consider the following Gröbner basis.

```
gap> p1 := [[[1,1,2],[]],[1,-1]];;
gap> p2 := [[[1,2,2],[]],[1,-1]];;
gap> G := Grobner([p1,p2]);;
gap> PrintNPList(G);
 b - a
 a^3 - 1
```

Print the product in the quotient algebra of the polynomials $a-2$ and $b-3$ by use of `MulQA`:

```
gap> s1 := [[[1],[]],[1,-2]];;
gap> s2 := [[[2],[]],[1,-3]];;
gap> PrintNP(MulQA(s1,s2,G));
 a^2 - 5a + 6
```

The result should be equal to the strong normal form of the product of $a-2$ and $b-3$ with respect to `G`:

```
gap> MulQA(s1,s2,G) = StrongNormalFormNP(MulNP(s1,s2),G);
true
```

3.5.6 StrongNormalFormNP

▷ `StrongNormalFormNP(f, G)` (function)

Returns: The strong normal form of a polynomial with respect to `G`

When invoked with a polynomial in NP format (see Section 2.1) and a finite set `G` of polynomials in NP format, this function will return a strong normal form (that is, a polynomial that is equal to `f` modulo `G`, every monomial of which is a multiple of no leading monomial of an element of `G`).

Note that the StrongNormalForm with respect to a Gröbner basis is uniquely determined, but that for an arbitrary input `G` the result may depend on the order in which the individual reduction steps are implemented.

Example: Consider the following two polynomials in NP format.

```
gap> p1 := [[[1,1,2],[]],[1,-1]];;
gap> p2 := [[[1,2,2],[]],[1,-1]];;
```

The strong normal form of the polynomial

```
gap> p := [[[1,1,1,2],[2,1],[]],[1,-1,3]];;
```

with respect to the list `[p1,p2]` is computed by use of the function `StrongNormalFormNP`:

```
gap> PrintNP(StrongNormalFormNP(p,[p1,p2]));
 - ba + a + 3
```

3.6 Finiteness and Hilbert series

3.6.1 DetermineGrowthQA

▷ `DetermineGrowthQA(`*`Lm, t, exact`*`)` (function)

Returns: If the quotient algebra is finite dimensional, then the integer `0` is returned. If the growth is polynomial and the algorithm found a precise degree `d` of the growth polynomial, then `d` is returned. If the growth is polynomial and no precise answer is found, an interval `[d1,d2]` is returned in which the dimension lies. If the growth is exponential, the string `"exponential growth"` is returned.

Given leading monomials *`Lm`* of some Gröbner basis (these can be obtained with the function `LMonsNP` (3.3.10)), the number *`t`* of generators of a free algebra, say A, in which the monomials lie, and a boolean *`exact`*, this function checks whether the quotient algebra of A by the ideal generated by *`Lm`* is finite dimensional. In doing so it constructs a graph of normal words which helps with the computations. It also checks for exponential or polynomial growth in the infinite case.

If the precise degree is needed in the polynomial case, the argument *`exact`* should be set to `true`.

The function `DetermineGrowthQA` allows preprocessing, which may speed up the computations. This can be done with the function `PreprocessAnalysisQA` (3.6.4).

Example: For the list of monomials consisting of a single variable in a free algebra generated by two variables the growth is clearly polynomial of degree 1. This is verified by invoking `DetermineGrowthQA` with arguments `[[1]]` (the list of the single monomial consisting of the first variable), the number of generators of the free algebra to which the monomials belong (which is 2 here), and the boolean `true` indicating that we wish a precise degree in case of polynomial growth.

```
gap> DetermineGrowthQA([[1]],2,true);
1
```

Here is an example of polynomial growth of degree 2:

```
gap> L := [[1,2,1],[2,2,1]];
[ [ 1, 2, 1 ], [ 2, 2, 1 ] ]
gap> DetermineGrowthQA(L,2,true);
2
```

In order to show how to apply the function to arbitrary polynomials, consider the following two polynomials in NP format.

```
gap> F := GF(256);
GF(2^8)
gap> z := GeneratorsOfField(F)[1];
Z(2^8)
gap> p1 := [[[1,1,1,2],[]],[z,1]];;
gap> p2 := [[[2,2,2,1],[]],[1,z]];;
```

The polynomials p1 and p2 have coefficients in the field F of order 256. In order to study the growth of the quotient algebra we first compute the list of leading monomials of the Gröbner basis elements and next apply DetermineGrowthQA.

```
gap> GB := Grobner([p1,p2]);;
gap> L := LMonsNP(GB);;
gap> for lm  in L  do PrintNP( [ [ lm ], [ 1 ] ] ); od;
 a^3b
 b^2
 ba
 a^5
gap> DetermineGrowthQA(L,2,true);
0
```

3.6.2 FinCheckQA

▷ FinCheckQA(*Lm, t*) (function)

Returns: true, if the quotient algebra is finite dimensional and false otherwise

Given a list *Lm* of leading monomials such that none of these divides another, and the number *t* of generators of a free algebra in which they are embedded, this function checks whether the quotient algebra of the free algebra by the ideal generated by *Lm* is finite dimensional.

When given a Gröbner basis *G*, the dimension of the quotient algebra of the free algebra by the ideal generated by *G* coincides with the the dimension of the quotient algebra of the free algebra by the ideal generated by the leading terms of elements of *G*. These can be obtained from *G* with the function LMonsNP (3.3.10).

The function FinCheckQA allows for preprocessing with the function PreprocessAnalysisQA (3.6.4). This may speed up the computation.

Example: Consider the following list L of two monomials.

```
gap> L := [[1,2,1],[2,2,1]];;
```

Finiteness of the dimension of the quotient algebra of the free algebra by the ideal generated by these two monomials can be decided by means of FinCheckQA. Its arguments are L and the number of generators of the free algebra in which the monomials reside.

```
gap> FinCheckQA(L,2);
false
```

This example turns out to be infinite dimensional. Here is a finite-dimensional example.

```
gap> FinCheckQA([[1],[2,2]],2);
true
```

3.6.3 HilbertSeriesQA

▷ HilbertSeriesQA(*Lm, t, d*) (function)

Returns: A list of coefficients of the Hilbert series up to degree *d*

Given a set of monomials *Lm*, none of which divides another, and the number *n* of generators of the free algebra in which they occur, this function computes the Hilbert series up to a given degree *d*.

Internally, it builds (part of) the graph of standard words. This function will remove zeroes from the end of the list of coefficients.

Example: Consider the following list L of two monomials.

```
gap> L := [[1,2,1],[2,2,1]];;
```

Finiteness of the dimension of the quotient algebra of the free algebra by the ideal generated by these two monomials can be decided by means of FinCheckQA. Its arguments are L and the number of generators of the free algebra in which the monomials reside.

```
gap> HilbertSeriesQA(L,2,10);
[ 1, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20 ]
```

This indicates that the growth may be polynomial. DetermineGrowthQA (3.6.1) can be used to check this.

3.6.4 PreprocessAnalysisQA

▷ PreprocessAnalysisQA(*Lm, t, iterations*) (function)

Returns: The left-reduced list of 'obstructions', obtained by applying left-reduction recursively

This preprocessing of the list *Lm* of monomials can be applied to the set of leading terms of a Gröbner basis before calling the functions FinCheckQA (3.6.2) or DetermineGrowthQA (3.6.1), in order to speed up calculations using these functions. As the name suggests, *t* should be the size of the alphabet. The parameter *iterations* gives the maximum number of recursion steps in the preprocessing (*0* means no restriction). For more information about this function see [Kro03].

Example: Consider the following two polynomials in NP format of which a Gröbner basis is computed.

```
F := GF(256);
z := GeneratorsOfField(F)[1];
gap> p1 := [[[1,1,1,1,2],[]],[1,-1]];;
gap> p2 := [[[2,2,2,1,1,1],[]],[1,-1]];;
gap> GB := Grobner([p1,p2]);;
gap> PrintNPList(GB);
 a^4b - 1
 ba - ab
 b^2 - a
 a^5 - b
```

Application of PreprocessAnalysisQA is carried out on the leading terms of GB, with 2, 4, 8, recursions, respectively.

```
gap> L := LMonsNP(GB);
[ [ 1, 1, 1, 1, 2 ], [ 2, 1 ], [ 2, 2 ], [ 1, 1, 1, 1, 1 ] ]
gap> L1 := PreprocessAnalysisQA(L,2,2);
[ [ 1, 1, 1 ], [ 2, 1 ], [ 1, 1, 2 ], [ 2, 2 ] ]
gap> L2 := PreprocessAnalysisQA(L1,2,4);
[ [ 1 ], [ 2 ] ]
```

3.7 Functions of the trace variant

3.7.1 EvalTrace

▷ `EvalTrace(`*p*`, `*Lnp*`)` (function)

Returns: The trace evaluated to a polynomial in NP format.

For a traced polynomial *p* and a list *Lnp* of polynomials in NP format, this program evaluates the trace by substituting the polynomials of *Lnp* back in the expression `p.trace` and computing the resulting polynomial. The result should have the same value as `p.pol`.

Example: First we compute the traced Gröbner basis of the list of the following two polynomials in NP format.

```
gap> p1 := [[[1,1,2],[]],[1,-1]];;
gap> p2 := [[[2,2,1],[]],[1,-1]];;
gap> Lnp := [p1,p2];;
gap> GBT := SGrobnerTrace(Lnp);;
```

In order to check that the polynomials in `GBT` belong to the ideal generated by `p1` and `p2`, we evaluate the trace. For each traced polynomial `p` in `GBT`, the polynomial `p.pol` is equated to the evaluated expression `p.trace`, in which each occurrence of `G(i)` is replaced by `Lnp[i]` by use of `EvalTrace`.

```
gap> ForAll(GBT,q -> EvalTrace(q,Lnp) = q.pol);
true
```

3.7.2 PrintTraceList

▷ `PrintTraceList(`*G*`)` (function)

When invoked with a list *G* of traced polynomials, this function prints the traces of that list.

Example: First we compute the traced Gröbner basis of the list of two polynomials in NP format and next we print it by use of `PrintTraceList`.

```
gap> p1 := [[[1,1,2],[]],[1,-1]];;
gap> p2 := [[[2,2,1],[]],[1,-1]];;
gap> GBT := SGrobnerTrace([p1,p2]);;
gap> PrintTraceList(GBT);
 aG(1) - bG(1) - G(1)ba^2b + a^2G(2)ab

 G(1)ba^2 + bG(1)ba + G(2) - a^2G(2)a - ba^2G(2)
```

3.7.3 PrintTracePol

▷ `PrintTracePol(`*p*`)` (function)

This function prints the trace of an NP polynomial *p*.

Example: First we compute the traced Gröbner basis of the list of two polynomials in NP format. Next we print the trace polynomial of the members of the list by use of `PrintTracePol`.

```
gap> p1 := [[[1,1,2],[]],[1,-1]];;
gap> p2 := [[[2,2,1],[]],[1,-1]];;
gap> GBT := SGrobnerTrace([p1,p2]);;
gap> for np in GBT do PrintTracePol(np); Print("\n"); od;
 aG(1) - bG(1) - G(1)ba^2b + a^2G(2)ab

 G(1)ba^2 + bG(1)ba + G(2) - a^2G(2)a - ba^2G(2)
```

3.7.4 PrintNPListTrace

▷ `PrintNPListTrace(`*G*`)` (function)

When invoked with a set of traced non-commutative polynomials *G*, this function prints the list of the traced polynomials, without the trace.

Example: First we compute the traced Gröbner basis of the list of two polynomials in NP format. Next we print the polynomials found by use of `PrintNPListTrace`.

```
gap> p1 := [[[1,1,2],[]],[1,-1]];;
gap> p2 := [[[2,2,1],[]],[1,-1]];;
gap> GBT := SGrobnerTrace([p1,p2]);;
gap> PrintNPListTrace(GBT);
 b - a
 a^3 - 1
```

3.7.5 SGrobnerTrace

▷ `SGrobnerTrace(`*Lnp*`)` (function)

Returns: Gröbner Basis, traceable

For a list of noncommutative polynomials *Lnp* this function will use Buchberger's algorithm with strong normal form to find a Gröbner Basis `G` (if possible; the general problem is unsolvable).

The results will be traceable. Functions that can print the Gröbner basis are `PrintTraceList` (3.7.2) (with the trace) and `PrintNPListTrace` (3.7.4) (without the trace).

Example: For the list of the following two polynomials in NP format, a traced Gröbner basis is computed.

```
gap> p1 := [[[1,1,2],[]],[1,-1]];;
gap> p2 := [[[2,2,1],[]],[1,-1]];;
gap> GBT := SGrobnerTrace([p1,p2]);
[ rec( pol := [ [ [ 2 ], [ 1 ] ], [ 1, -1 ] ],
      trace := [ [ [  ], 1, [ 2, 1, 1, 2 ], -1 ], [ [ 2 ], 1, [  ], -1 ],
          [ [ 1 ], 1, [  ], 1 ], [ [ 1, 1 ], 2, [ 1, 2 ], 1 ] ] ),
  rec( pol := [ [ [ 1, 1, 1 ], [  ] ], [ 1, -1 ] ],
      trace := [ [ [ 2 ], 1, [ 2, 1 ], 1 ], [ [  ], 1, [ 2, 1, 1 ], 1 ],
          [ [  ], 2, [  ], 1 ], [ [ 2, 1, 1 ], 2, [  ], -1 ],
          [ [ 1, 1 ], 2, [ 1 ], -1 ] ] ) ]
```

3.7.6 StrongNormalFormTraceDiff

▷ `StrongNormalFormTraceDiff(`*np, GBT*`)` (function)

Returns: The traced polynomial for the difference of *f* with the strong normal form of *np* with respect to *GBT*

When invoked with a polynomial *np* in NP format as its first argument, and a traced Gröbner basis *GBT* as generated by `SGrobnerTrace` (3.7.5), this function returns the difference of *np* with the strong normal form of *np* with respect to *GBT*. This difference `d` is returned as a traced polynomial. The trace information `d.trace` gives an expression of `d.pol` as a combination of polynomials from the list of polynomials to which the trace parts of *GBT* are referring. Typically, this is the set of relations used as input to the computation of *GBT*.

Note that the difference of the polynomials *np* and `d.pol` is the same as the StrongNormalForm of *np*.

Example: First we compute the traced Gröbner basis of the list of the following two polynomials in NP format.

```
gap> p1 := [[[1,1,2],[]],[1,-1]];;
gap> p2 := [[[2,2,1],[]],[1,-1]];;
gap> GBT := SGrobnerTrace([p1,p2]);;
```

Of the polynomial a^6 we compute its difference with the normal form. The result is printed by used of `PrintNP` (3.2.1) and `PrintTraceList` (3.7.2).

```
gap> f := [[[1,1,1,1,1,1]],[1]];;
gap> sf := StrongNormalFormTraceDiff(f,GBT);;
gap> PrintNP(sf.pol);
 a^6 - 1
gap> PrintTraceList([sf]);
 G(1)ba^2 + bG(1)ba + G(1)ba^5 + bG(1)ba^4 + G(2) + G(2)a^3 - a^2G(
2)a - ba^2G(2) - a^2G(2)a^4 - ba^2G(2)a^3
```

3.8 Functions of the truncated variant

3.8.1 Examples

More about these functions can be found in A.4

3.8.2 SGrobnerTrunc

▷ `SGrobnerTrunc(`*Lnp, deg, wtv*`)` (function)

Returns: A list of homogeneous NP polynomials if the first argument of the input is a list of homogeneous NP polynomials, and the boolean `false` otherwise.

This functions should be invoked with a list *Lnp* of polynomials in NP format, a natural number *deg*, and a weight vector *wtv* of length the number of generators of the free algebra A containing the elements of *Lnp*, and with positive integers for entries. If the polynomials of *Lnp* are homogeneous with respect to *wtv*, the function will return a Gröbner basis of *Lnp* truncated above *deg*. If the list of polynomials *Lnp* is not homogeneous with respect to *wtv*, it returns `false`. The homogeneity check can be carried out by `CheckHomogeneousNPs` (3.8.3).

Example: Consider the following two polynomials in NP format.

```
gap> p1 := [[[1,2,2,1],[2,1,1,2]],[1,-1]];;
gap> p2 := [[[2,2,2],[1,1]],[1,-1]];;
gap> PrintNPList([p1,p2]);
 ab^2a - ba^2b
 b^3 - a^2
```

These are homogeneous with respect to weights $[3,2]$. The degrees are 10 and 6, respectively. The Gröbner basis truncated above degree 12 of the list `[p1,p2]` is computed and subsequently printed as follows.

```
gap> PrintNPList(SGrobnerTrunc([p1,p2],12,[3,2]));
 ba^2 - a^2b
 b^3 - a^2
 ab^2a - a^2b^2
```

3.8.3 CheckHomogeneousNPs

▷ `CheckHomogeneousNPs(`*`Lnp, wtv`*`)` (function)

Returns: A list of weighted degrees of the polynomials if these are homogeneous with respect to *`wtv`*, and `false` otherwise.

When invoked with a list of NP polynomials *`Lnp`* and a weight vector *`wtv`* (whose entries should be positive integers), this function returns the list of weighted degrees of the polynomials in *`Lnp`* if these are all homogeneous and nonzero, and `false` otherwise. Here, a polynomial is (weighted) homogeneous with respect to a weight vector w if there is constant d such that, for each monomial $[t_1,\ldots,t_r]$ of the polynomial the sum of all $w[t_i]$ for i in $[1..r]$ is equal to d. The natural number d is then called the weighted degree of the polynomial.

Example: Consider the following two polynomials in NP format.

```
gap> p1 := [[[1,2,2,1],[2,1,1,2]],[1,-1]];;
gap> p2 := [[[2,2,2],[1,1]],[1,-1]];;
gap> PrintNPList([p1,p2]);
 ab^2a - ba^2b
 b^3 - a^2
```

These are homogeneous with respect to weights $[3,2]$. The degrees are 10 and 6, respectively. This is checked as follows.

```
gap> CheckHomogeneousNPs([p1,p2],[3,2]);
[ 10, 6 ]
```

3.8.4 BaseQATrunc

▷ `BaseQATrunc(`*`Lnp, deg, wtv`*`)` (function)

Returns: A list of monomials if the first argument of the input is a list of homogeneous NP polynomials or `false`.

When invoked with a list of polynomials *`Lnp`*, a natural number *`deg`*, and a weight vector *`wtv`* of length the number of variables and with positive integers for entries, such that the polynomials of *`Lnp`* are homogeneous with respect to *`wtv`*, it returns a list whose i-th entry is a basis of monomials of

the homogeneous part of degree $i-1$ the quotient algebra of the free noncommutative algebra by the weighted homogeneous ideal generated by *Lnp* truncated above *deg*. If the list of polynomials *Lnp* is not homogeneous, it returns `false`.

Example: Consider the truncated Gröbner basis of the following two polynomials in NP format.

```
gap> p1 := [[[1,2,2,1],[2,1,1,2]],[1,-1]];;
gap> p2 := [[[2,2,2],[1,1]],[1,-1]];;
gap> wtv := [3,2];;
gap> GB := SGrobnerTrunc([p1,p2],12,wtv);;
gap> GBNP.ConfigPrint("a","b");
gap> PrintNPList(GB);
 ba^2 - a^2b
 b^3 - a^2
 ab^2a - a^2b^2
```

A basis of standard monomials is found and printed as follows.

```
gap> BT := BaseQATrunc(GB,12,wtv);;
gap> for degpart in BT do
>   for mon in degpart do PrintNP([[mon],[1]]); od;
> od;
 1
 b
 a
 b^2
 ba
 ab
 a^2
 b^3
 b^2a
 bab
 ab^2
 aba
 a^2b
 b^4
 a^3
 b^3a
 b^2ab
 bab^2
 ab^3
 baba
 abab
 a^2b^2
 b^5
 a^2ba
 b^4a
 a^3b
 b^3ab
 b^2ab^2
 bab^3
 ab^4
 a^4
```

```
b^2aba
ab^3a
babab
abab^2
a^2b^3
b^6
```

3.8.5 DimsQATrunc

▷ `DimsQATrunc(`*`Lnp, deg, wtv`*`)` (function)

Returns: A list of monomials if the first argument of the input is a list of homogeneous NP polynomials or `false`.

When invoked with a list of polynomials *Lnp*, a natural number *deg*, and a weight vector *wtv* of length the number of variables and with positive integers for entries, such that the polynomials of *Lnp* are homogeneous with respect to *wtv*, it returns a list of dimensions of the homogeneous parts of the quotient algebra of the free noncommutative algebra by the ideal generated by *Lnp* truncated above *deg*. The i-th entry of the list gives the dimension of the homogeneous part of degree $i-1$ of the quotient algebra. If the list of polynomials *Lnp* is not homogeneous, it returns `false`.

Example: Consider the truncated Gröbner basis of the following two polynomials in NP format.

```
gap> p1 := [[[1,2,2,1],[2,1,1,2]],[1,-1]];;
gap> p2 := [[[2,2,2],[1,1]],[1,-1]];;
gap> wtv := [3,2];;
gap> GB := SGrobnerTrunc([p1,p2],12,wtv);;
```

Information on the dimensions of the homogeneous parts of the quotient algebra is found as follows,

```
gap> DimsQATrunc(GB,12,wtv);
[ 1, 0, 1, 1, 1, 2, 2, 3, 3, 5, 4, 7, 7 ]
```

3.8.6 FreqsQATrunc

▷ `FreqsQATrunc(`*`Lnp, deg, wtv`*`)` (function)

Returns: A list of multiplicities of frequencies of monomials if the first argument of the input is a list of homogeneous polynomials in NP format, and `false` otherwise.

The frequency of a monomial is the list of numbers of occurrences of a variable in the monomial for each variable; the multiplicity of a frequency is the number of monomials in the standard basis for a quotient algebra with this frequency. When invoked with a list *Lnp* of polynomials in NP format representing a (truncated) Gröbner basis, a natural number *deg*, and a weight vector *wtv* of length the number of variables and with positive integers for entries, such that the polynomials of *Lnp* are homogeneous with respect to *wtv*, it returns a list of frequencies occurring with their multiplicities for the quotient algebra of the free noncommutative algebra by the ideal generated by *Lnp* truncated above *deg*. The i-th entry of the list gives the frequencies of the weight $(i-1)$ basis monomials of the quotient algebra. If the list of polynomials *Lnp* is not homogeneous with respect to *wtv*, it returns `false`.

Example: Consider the truncated Gröbner basis of the following two polynomials in NP format.

```
gap> p1 := [[[1,2,2,1],[2,1,1,2]],[1,-1]];;
gap> p2 := [[[2,2,2],[1,1]],[1,-1]];;
gap> wtv := [3,2];;
gap> GB := SGrobnerTrunc([p1,p2],12,wtv);;
gap> PrintNPList(GB);
 ba^2 - a^2b
 b^3 - a^2
 ab^2a - a^2b^2
```

The multiplicities of the frequencies of of monomials in a standard basis of the quotient algebra with respect to the ideal generated by GB is found as follows, for weights up to and including 8.

```
gap> F := FreqsQATrunc(GB,8,wtv);
[ [ [ [  ], 1 ] ], [ [ [ 0, 1 ], 1 ] ], [ [ [ 1, 0 ], 1 ] ],
  [ [ [ 0, 2 ], 1 ] ], [ [ [ 1, 1 ], 2 ] ],
  [ [ [ 2, 0 ], 1 ], [ [ 0, 3 ], 1 ] ], [ [ [ 1, 2 ], 3 ] ],
  [ [ [ 2, 1 ], 2 ], [ [ 0, 4 ], 1 ] ] ]
```

The interpretation of this data is given by the following lines of code.

```
gap> for f in F do
>   if f[1][1] <> [] then
>     Print("At level ", wtv * (f[1][1]), " the multiplicities are\n");
>     for x in f do
>       Print("  for ",x[1],": ",x[2],"\n");
>     od;
>   else
>     Print("At level ", 0 , " the multiplicity of [] is ",f[1][2],"\n");
>   fi;
>   Print("\n");
> od;
At level 0 the multiplicity of [] is 1

At level 2 the multiplicities are
  for [ 0, 1 ]: 1

At level 3 the multiplicities are
  for [ 1, 0 ]: 1

At level 4 the multiplicities are
  for [ 0, 2 ]: 1

At level 5 the multiplicities are
  for [ 1, 1 ]: 2

At level 6 the multiplicities are
  for [ 2, 0 ]: 1
  for [ 0, 3 ]: 1

At level 7 the multiplicities are
  for [ 1, 2 ]: 3
```

```
At level 8 the multiplicities are
  for [ 2, 1 ]: 2
  for [ 0, 4 ]: 1
```

3.9 Functions of the module variant

3.9.1 SGrobnerModule

▷ SGrobnerModule(*Lnpm, Lnp*) (function)

Returns: A record GBR containing a Gröbner basis (if found...the general problem is unsolvable) for modules; GBR.p will contain the prefix rules and GBR.ts will contain the two-sided rules, and GBR.pg will be the smallest rank of the free module to which all prefix relations belong

For a list *Lnpm* of vectors in NPM format (see Section 2.1), and a list *Lnp* of polynomials in NP format, this function will use Buchberger's algorithm with strong normal form applied to the union of *Lnpm*, *Lnp*, the set of polynomials $x*e-x$ and $x*m[i]$ for x a standard indeterminate, a module generator $m[j]$ or the dummy indeterminate e, and the set of all $e*x-x$ for x a standard indeterminate, to find a Gröbner Basis record GBR (if possible; the general problem is unsolvable). This record will have a Gröbner Basis GBR.ts for the two-sided ideal generated by *Lnp* and an intersection with the module GBR.p representing the module relations needed to find representative vectors in the module uniquely by means of a strong normal form computation modding out GBR.p and, for the scalars, GBR.ts.

Example: Consider the following two polynomials in NP format.

```
gap> p1 := [[[1,1,2],[]],[1,-2]];;
gap> p2 := [[[1,2,2],[]],[1,-3]];;
```

Consider also the following two vectors in NPM format.

```
gap> v1 := [[[-1,1,2],[-1]],[1,-1]];;
gap> v2 := [[[-2,2,2],[-2]],[1,-2]];;
```

The Gröbner basis record for this data is found by SGrobnerModule:

```
gap> GBR := SGrobnerModule([v1,v2],[p1,p2]);;
```

The record GBR has two fields, p for prefix relations (vectors in the module) and ts for two-sided relations (polynomials in the algebra):

```
gap> PrintNPList(GBR.p);
[ 0, 1 ]
[ 1 , 0]
gap> PrintNPList(GBR.ts);
 b - 3/2a
 a^3 - 4/3
```

3.9.2 BaseQM

▷ BaseQM(*GBR, t, mt, maxno*) (function)

Returns: A basis of the module obtained from the free module of rank *mt* over the free algebra on *t* generators by factoring out the submodule generated by the elements of *GBR*

When called with a Gröbner basis record *GBR* (see Section 2.8), the number of variables *t*, the number of module generators *mt*, and a maximum number of terms to be found, *maxno*, the function BaseQM will return a (partial) base of the quotient module of A^{mt} over the free algebra on A on t generators by the right sub A-module generated by the elements of *GBR*. Note that the record *GBR* consists of two fields: the list *GBR.p* of vectors in NPM format representing elements of the free module A^{mt}, and the list *GBR.ts* of polynomials in NP format representing elements of A. The submodule generated by *GBR* is considered to be the right submodule of A^{mt} generated by *GBR.p* and all elements of the form $v \cdot np$ with np in the two-sided ideal of A generated by *GBR.ts* and v in A^{mt}. If this function is invoked with *maxno* equal to 0, then a full basis will be given.

If $t = 0$, then t will be set to the minimal value such that all polynomials of *GBR.ts* and all polynomials occurring in *GBR.p* have at most t variables.

If $mt = 0$, then mt will be set to the minimal value such that all vectors of *GBR.p* belong to A^{mt}.

If the module is cyclic (that is, has a single generator), it is possible to use the Gröbner basis of the ideal in the algebra instead of the Gröbner basis record. This can be done by entering 0 for the number *mt* of module generators.

Example: Consider the following two polynomials in NP format.

```
gap> p1 := [[[1,1,1,2],[]],[1,-1]];;
gap> p2 := [[[2,2,2,1],[]],[1,-1]];;
```

Consider also the following two vectors in NPM format.

```
gap> v1 := [[[-1,1,2],[-1]],[1,-1]];;
gap> v2 := [[[-2,2,2],[-2]],[1,-2]];;
```

The Gröbner basis record for this data is found by SGrobnerModule (3.9.1):

```
gap> GBR := SGrobnerModule([v1,v2],[p1,p2]);;
gap> PrintNPList(GBR.ts);
 ba - ab
 b^2 - a^2
 a^3b - 1
 a^5 - b
gap> PrintNPList(GBR.p);
[ 0, 1 ]
[ b - a , 0]
[ a^2 - 1 , 0]
[ ab - 1 , 0]
```

The function BaseQM computes a basis.

```
gap> B := BaseQM(GBR,2,2,0);;
gap> PrintNPList(B);
[ 1 , 0]
[ a , 0]
```

The function BaseQM with arguments so as to let the number of dimensions of the module and the number of variables be chosen minimal.

```
gap> B := BaseQM(GBR,0,0,0);;
gap> PrintNPList(B);
[ 1 , 0]
[ a , 0]
```

The function BaseQM can also be used to ompute the first three elements of a basis.

```
gap> B := BaseQM(GBR,2,2,3);;
gap> PrintNPList(B);
[ 1 , 0]
[ a , 0]
```

3.9.3 DimQM

▷ DimQM(*GBR*, *t*, *mt*) (function)

Returns: The dimension of the quotient module

When called with a Gröbner basis record *GBR* (see Section 2.8), a number of variables *t* at least equal to the number of generators involved in the polynomials of *GBR.p* and *GBR.ts*, and a number of generators *mt* of a free module containing the prefix relations in *GBR.p*, the function DimQM will return the dimension over the coefficient field of the quotient module of the free right module A^{mt} of rank *mt* over the free algebra A on t generators by the right sub A-module generated by the elements of *GBR*, if this dimension is finite. Otherwise, the computation invoked by the function will not terminate.

If $t = 0$, then t will be set to the minimal value such that all polynomials of *GBR.ts* and all polynomials occurring in *GBR.p* belong to A^{mt}.

If $mt = 0$, then mt will be set to the minimal value such that all vectors of *GBR.p* belong to A^{mt}.

Example: Consider the following two polynomials in NP format.

```
gap> p1 := [[[1,1,1,2],[]],[1,-1]];;
gap> p2 := [[[2,2,2,1],[]],[1,-1]];;
```

Consider also the following two vectors in NPM format.

```
gap> v1 := [[[-1,1,2],[-2]],[1,-1]];;
gap> v2 := [[[-2,2,2],[-1]],[1,-2]];;
```

The Gröbner basis record for this data is found by SGrobnerModule (3.9.1):

```
gap> GBR := SGrobnerModule([v1,v2],[p1,p2]);;
```

The function DimQM computes the dimension over the rationals of the quotient of the free module over the free algebra on two generators by the submodule generated by the vectors v1, v2, $[p,q]$, where p and q run over all elements of the two-sided ideal in the free algebra generated by p1 and p2.

```
gap> SetInfoLevel(InfoGBNP,2);
gap> DimQM(GBR,2,2);
0
```

The answer should be equal to the size of `BaseQM(GBR,t,mt,0)`.

```
gap> DimQM(GBR,2,2) = Length(BaseQM(GBR,2,2,0));
true
gap> SetInfoLevel(InfoGBNP,0);
```

3.9.4 MulQM

▷ `MulQM(`*p1, p2, GBR*`)` (function)

Returns: The strong normal form of the product *p1*∗*p2* with respect to *GBR*

When called with three arguments, the first of which, *p1*, is a module element in NPM format, the second of which, *p2*, is a polynomial in NP format representing an element of the quotient algebra, and the third of which is a Gröbner basis record *GBR*, this function will return the product `p1*p2` in the module.

Example: Consider the following two polynomials in NP format.

```
gap> p1 := [[[1,1,2],[]],[1,-1]];;
gap> p2 := [[[1,2,2],[]],[1,-1]];;
gap> PrintNPList([p1,p2]);
 a^2b - 1
 ab^2 - 1
```

Consider also the following two vectors in NPM format.

```
gap> v1 := [[[-1,1,2],[-1]],[1,-1]];;
gap> v2 := [[[-2,2,2],[-2]],[1,-2]];;
gap> PrintNPList([v1,v2]);
[ ab - 1 , 0]
[ 0, b^2 - 2 ]
```

The Gröbner basis record for this data is found by `SGrobnerModule` (3.9.1):

```
gap> GBR := SGrobnerModule([v1,v2],[p1,p2]);;
gap> PrintNPList(GBR.ts);
 b - a
 a^3 - 1
gap> PrintNPList(GBR.p);
[ 0, 1 ]
[ a - 1 , 0]
```

The function `MulQM` computes the product of the vector `w` with the polynomial `q`.

```
gap> w := [[[-1,2],[-2,1]],[1,-4]];;
gap> PrintNP(w);
[ b , - 4a ]
gap> q := [[[2,2,1],[1]],[2,3]];;
gap> PrintNP(q);
 2b^2a + 3a
gap> wq := MulQM(w,q,GBR);;
gap> PrintNP(wq);
[ 5 , 0]
```

3.9.5 StrongNormalFormNPM

▷ `StrongNormalFormNPM(f, GBR)` (function)

Returns: The strong normal form of a polynomial in NP format with respect to *GBR*

When invoked with a polynomial in NP format (see Section 2.1) and a Gröbner basis record *GBR* (see Section 2.8), this function will return the strong normal form (the polynomial reduced by the prefix and two-sided relations of the Gröbner basis combination).

This function assumes that *GBR.p* and *GBR.ts* are ordered (with the ordering `LtNP` (3.3.9)), that the polynomials in *GBR.ts* are monic and clean (see `MkMonicNP` (3.3.12) and `CleanNP` (3.3.7)), and that the polynomial *f* is clean. Note that a Gröbner basis record as returned by a function like `SGrobnerModule` (3.9.1) is in the required form.

Example: Consider the following two polynomials in NP format.

```
gap> p1 := [[[1,1,2],[]],[1,-1]];;
gap> p2 := [[[1,2,2],[]],[1,-1]];;
gap> PrintNPList([p1,p2]);
 a^2b - 1
 ab^2 - 1
```

Consider also the following two vectors in NPM format.

```
gap> v1 := [[[-1,1,2],[-1]],[1,-1]];;
gap> v2 := [[[-2,2,2],[-2]],[1,-2]];;
gap> PrintNPList([v1,v2]);
[ ab - 1 , 0]
[ 0, b^2 - 2 ]
```

The Gröbner basis record for this data is found by `SGrobnerModule` (3.9.1):

```
gap> GBR := SGrobnerModule([v1,v2],[p1,p2]);;
gap> PrintNPList(GBR.ts);
 b - a
 a^3 - 1
gap> PrintNPList(GBR.p);
[ 0, 1 ]
[ a - 1 , 0]
```

The vector `w` is brought into strong normal form with respect to `GBR`:

```
gap> w := [[[-1,2],[-2,1]],[1,-4]];;
gap> PrintNP(w);
[ b , - 4a ]
gap> v := StrongNormalFormNPM(w,GBR);;
gap> PrintNP(v);
[ 1 , 0]
```

Chapter 4

Info Level

4.1 Introduction

Many functions of the GBNP package can produce additional output. Such output might be useful for long calculations, to see where the calculation is or to gain more information about the calculation itself.

GAP provides the tools to be able to tune the output of the functions. All of the functions of this package use the `InfoClass InfoGBNP` (4.2.1) and some use the `InfoClass InfoGBNPTime` (4.3.1). As usual with GAP when this is left at 0, the functions will hardly print additional information. It can be set to 1 or 2 with `SetInfoLevel` (more about this function can be found at **Reference: InfoLevel**). A brief explanation about each infolevel will be given in the next sections.

4.2 InfoGBNP

4.2.1 InfoGBNP

▷ `InfoGBNP` (info class)

The `InfoClass` for this package is used in almost all functions. To change this level to 1 (some information) or 2 (more information, also information from calculation loops) use the function `SetInfoLevel`.

4.2.2 What will be printed at level 0

At level 0 no information is printed beyond what functions themselves command to be printed. These include functions like `PrintNP` (3.2.1), `PrintNPList` (3.2.3), `PrintTraceList` (3.7.2) and `PrintNPListTrace` (3.7.4), but it also includes the function `DetermineGrowthQA` (3.6.1), which only prints one or two lines and `DimsQATrunc` (3.8.5) which produces information about a truncated Gröbner basis.

4.2.3 What will be printed at level 1

The infolevel can be set to 1 with the following command:

```
SetInfoLevel(InfoGBNP,1);
```

At level 1 a large set of functions will produce a bit of output. Most of this information reports on the phase of the algorithm the calculations are in or some simple statistics about the input or output.

4.2.4 What will be printed at level 2

The infolevel can be set to 2 with the following command.

```
SetInfoLevel(InfoGBNP,2);
```

At level 2 a large set of functions will produce a lot of output. This mostly concerns information on loops in the calculations. Timing information will be printed as well.

4.3 InfoGBNPTime

4.3.1 InfoGBNPTime

▷ `InfoGBNPTime` (info class)

The `InfoClass` for timing is used in producing information about the runtime of the algorithm in certain possibly lengthy calculations.

To change this level to 1 (Gröbner functions give information) or 2 (more information, also information from other functions, which might not always take a long time and from inside loops) use the function `SetInfoLevel`.

4.3.2 What will be printed at level 0

No timing information will be printed at level 0. This can be desirable for small examples or when producing test output, for use with `Test`.

4.3.3 What will be printed at level 1

The infolevel can be set to 1 with the following command:

```
SetInfoLevel(InfoGBNPTime,1);
```

At level one there will be time information printed by the functions from different variants of the Gröbner basis algorithm: `Grobner` (3.4.1), `SGrobner` (3.4.2), `SGrobnerTrace` (3.7.5), and `SGrobnerTrunc` (3.8.2).

4.3.4 What will be printed at level 2

The infolevel can be set to 2 with the following command:

```
SetInfoLevel(InfoGBNPTime,2);
```

At level two there will also be some information printed from a loop from within `SGrobnerTrunc` (3.8.2).

Chapter 5

NMO Manual

5.1 Introduction

Up until September 2023 the manual for Randall Cone's package NMO was provided by GBNP as a separate `manual.pdf`. The NMO manual now forms this chapter in the GBNP manual.

What follows is a description of the largely experimental project of providing arbitrary monomial orderings to the GBNP package. The addition of the orderings comes in the form of a library, and a patch to GBNP; the patching process being called at the GBNP user's discretion.

More precisely, after a user creates a monomial ordering via the NMO library functions, a routine is called which overwrites the two GBNP functions "LtNP" and "GtNP". In GBNP, these latter two functions are explicitly length-lexicographic monomial comparison functions, and are used in GBNP's Gröbner Basis routines. Therefore NMO allows for the creation of arbitrary monomial ordering comparison functions, which, after the patching process, will be used by GBNP in place of its native comparison functions.

NMO is an acronym for Noncommutative Monomial Orderings. Such orderings play a key role in research surrounding noncommutative Gröbner basis theory; see [Gre99], [Mor94]. This package is geared primarily toward the use and study of noncommutative (associative) free algebras with identity, over computational fields. We have done our best to write code that treats a more general class of algebras, but the routines have not been as extensively tested in those cases. Users of the package are encouraged to provide constructive feedback about this issue or any others; we have open ears to ways to better these research tools.

Flexibility in the creation and use of noncommutative monomial orderings has been our guiding principle in writing NMO. For example, two (or more) orderings can be chained together to form new orderings. It should be noted, however, that efficiency has also been considered in the design of NMO for commonly used monomial orderings for noncommutative rings (e.g. length left-lexicographic). That is to say, some monomial orderings that occur regularly in the study of noncommutative algebras have already been included in NMO.

Throughout this chapter, methods and functions are generally classed as *External* and *Internal* routines. *External* routines are methods and functions that will be most useful to the average user, and generally work directly with native GAP algebraic objects. *Internal* routines usually concern backend operations and mechanisms, and are often related to operations involving *NP representations* of GAP algebraic elements, or they are related to attributes of monomial orderings. Many examples of basic code use are provided; with some examples following the reference material for the functions or methods involved.

ACKNOWLEDGEMENTS

- Our immense gratitude to the authors of GBNP for allowing us to make a small contribution.
- Equal gratitude to Dr. Ed Green for his help as mentor and advisor, in both this project and many others.

5.2 NMO Files within GBNP

Per the GAP package standard, NMO library code is read in via the file `gbnp/read.g`. The following gives brief descriptions of each of the files loaded by `gbnp/read.g`, all of which reside in the `gbnp/lib/nmo/` subdirectory:

- `ncalgebra.gd`

  Sets up some nice categories and filters in GAP.

- `ncordmachine.g*`

  Code for creating the new GAP family of noncommutative monomial orderings, as well as its attending (internal) machinery.

- `ncorderings.g*`

  Sets up actual noncommutative monomial orderings. This is where some specific example routines for monomial orderings are included. The less-than functions determining monomial orderings should be collected here, e.g. the length left-lexicographic ordering is here.

- `ncinterface.g*`

  These files provide the interface to comparison routines for determining equivalence, less-than, and greater-than comparison between two algebraic elements under a given NMO ordering.

- `ncutils.g*`

  Helpful utility routines for patching and unpatching GBNP for use with an NMO ordering.

There is a documentation directory in `gbnp/doc/nmo` wherein the GAPDoc source for this chapter may be found.

Finally, there is an examples directory in `gbnp/doc/examples/nmo` where the plain GAP source can be found for the examples in the Quickstart section of this chapter.

5.3 Quickstart

This Quickstart assumes you've already installed the GBNP package in its proper home. If that's yet to be done, please see the GBNP package manual for installation instructions.

If the user wishes, cutting and pasting the commands which directly follow the GAP prompt `gap>` is a good way to become familiar with NMO via the examples below. Alternatively, code for the following examples may be found in `gbnp/doc/examples/nmo/example0*.g`.

This Quickstart covers specific use of the NMO package's functionality as pertaining to computing noncommutative Gröbner bases for various examples. There are NMO user-level routines beyond these Gröbner basis applications that may be of interest, all of which are documented in later sections.

5.3.1 NMO Example 1

Example 1 is taken from Dr. Edward Green's paper "Noncommutative Gröbner Bases, and Projective Resolutions", and is referenced as "Example 2.7" there; please see [Gre99] for more information.

Load the GBNP package with:

Example

```
gap> LoadPackage("gbnp", false);
true
gap> # remove any previous orderings
gap> UnpatchGBNP();
LtNP restored
GtNP restored
```

Create a noncommutative free algebra on 4 generators over the Rationals in GAP:

```
gap> A := FreeAssociativeAlgebraWithOne(Rationals,"a","b","c","d");
<algebra-with-one over Rationals, with 4 generators>
```

Label the generators of the algebra:

Example

```
gap> a := A.a; b := A.b; c := A.c; d := A.d;
(1)*a
(1)*b
(1)*c
(1)*d
```

Set up our polynomials, and convert them to NP format:

Example

```
gap> polys := [c*d*a*b-c*b, b*c-d*a];
[ (-1)*c*b+(1)*c*d*a*b, (1)*b*c+(-1)*d*a ]
gap> reps := GP2NPList( polys );
[ [ [ [ 3, 4, 1, 2 ], [ 3, 2 ] ], [ 1, -1 ] ],
          [ [ [ 4, 1 ], [ 2, 3 ] ], [ -1, 1 ] ] ]
```

Compute the Gröbner basis via GBNP using its default (length left-lexicographic) ordering; that is, without patching GBNP with an NMO ordering:

Example

```
gap> gbreps := Grobner( reps );;
gap> gb := NP2GPList( gbreps, A );
[ (1)*d*a+(-1)*b*c, (1)*(c*b)^2+(-1)*c*b ]
```

Create a (length left-lexicographic ordering, with generators ordered: a < b < c < d. Note: this is the default ordering of generators by NMO, if none is provided:

Example

```
gap> ml := NCMonomialLeftLengthLexOrdering(A);
NCMonomialLeftLengthLexicographicOrdering([ (1)*a, (1)*b, (1)*c, (1)*d ])
```

Patch GBNP with the ordering `ml`, and then run the same example. We should get the same answer as above:

Example
```
gap> PatchGBNP( ml );
LtNP patched.
GtNP patched.
gap> gbreps := Grobner( reps );;
gap> gb := NP2GPList( gbreps, A );
[ (1)*d*a+(-1)*b*c, (1)*(c*b)^2+(-1)*c*b ]
```

Now create a Length-Lexicographic ordering on the generators such that d < c < b < a:

Example
```
gap> ml2 := NCMonomialLeftLengthLexOrdering( A, [4,3,2,1] );
NCMonomialLeftLengthLexicographicOrdering([ (1)*d, (1)*c, (1)*b, (1)*a ])
```

Compute the Gröbner basis with respect to this new ordering on the same algebra:

Example
```
gap> PatchGBNP(ml2);
LtNP patched.
GtNP patched.
gap> gbreps2 := SGrobner( reps );;
gap> gb2 := NP2GPList( gbreps2, A );
[ (1)*b*c+(-1)*d*a, (1)*c*d*a*b+(-1)*c*b, (1)*(d*a)^2*b+(-1)*d*a*b,
  (1)*c*(d*a)^2+(-1)*c*d*a, (1)*(d*a)^3+(-1)*(d*a)^2 ]
```

5.3.2 NMO Example 2

This example is the same as Example 1 above, except that the length and left-lexicographic orderings are created independently and then chained to form the usual length left-lexicographic ordering. Hence, all results should be the same.

Remove any previous orderings. Create a noncommutative free algebra on 4 generators over the Rationals, label, and set up the example:

Example
```
gap> UnpatchGBNP();
LtNP restored
GtNP restored
gap> A := FreeAssociativeAlgebraWithOne(Rationals,"a","b","c","d");;
gap> a := A.a;; b := A.b;; c := A.c;; d := A.d;;
gap> polys := [ c*d*a*b-c*b, b*c-d*a ];;
gap> reps := GP2NPList( polys );;
```

Create left-lexicographic ordering with a < b < c < d:

Example
```
gap> lexord := NCMonomialLeftLexicographicOrdering( A );
NCMonomialLeftLexicographicOrdering([ (1)*a, (1)*b, (1)*c, (1)*d ])
```

Create a length ordering on monomials in *A*, with ties broken by the lexicographic order `lexord`:

Example
```
gap> lenlex := NCMonomialLengthOrdering( A, lexord );
NCMonomialLengthOrdering([ (1)*a, (1)*b, (1)*c, (1)*d ])
```

Patch GBNP and proceed with our example:

Example
```
gap> PatchGBNP( lenlex );;
LtNP patched.
GtNP patched.
gap> gbreps := Grobner( reps );;
gap> gb := NP2GPList( gbreps, A );
[ (1)*d*a+(-1)*b*c, (1)*(c*b)^2+(-1)*c*b ]
```

Now, proceed similarly, with the lexicographic order such that d < c < b < a:

Example
```
gap> lexord2 := NCMonomialLeftLexicographicOrdering( A, [4,3,2,1] );
NCMonomialLeftLexicographicOrdering([ (1)*d, (1)*c, (1)*b, (1)*a ])
gap> lenlex2 := NCMonomialLengthOrdering( A, lexord2 );
NCMonomialLengthOrdering([ (1)*a, (1)*b, (1)*c, (1)*d ])
gap> PatchGBNP( lenlex2 );;
LtNP patched.
GtNP patched.
gap> gbreps2 := Grobner( reps );;
gap> gb2 := NP2GPList( gbreps2, A );
[ (1)*b*c+(-1)*d*a, (1)*c*d*a*b+(-1)*c*b, (1)*(d*a)^2*b+(-1)*d*a*b,
  (1)*c*(d*a)^2+(-1)*c*d*a, (1)*(d*a)^3+(-1)*(d*a)^2 ]
```

An important point can be made here. Notice that when the `lenlex2` length ordering is created, a lexicographic (generator) ordering table is assigned internally to the ordering since one was not provided to it. This is merely a convenience for lexicographically-dependent orderings, and in the case of the length order, it is not used. Only the lex table for `lexord2` is ever used. Some clarification may be provided in examining:

Example
```
gap> HasNextOrdering( lenlex2 );
true
gap> NextOrdering( lenlex2 );
NCMonomialLeftLexicographicOrdering([ (1)*d, (1)*c, (1)*b, (1)*a ])
gap> LexicographicTable( NextOrdering( lenlex2 ) );
[ (1)*d, (1)*c, (1)*b, (1)*a ]
```

5.3.3 NMO Example 3

Example 3 is taken from the book “Ideals, Varieties, and Algorithms”, ([CLO97], Example 2, p. 93-94); it is a commutative example.

First, set up the problem and find a Gröbner basis with respect to the length left-lexicographic ordering implicitly assumed in GBNP, after removing any previous orderings:

Example
```
gap> UnpatchGBNP();
LtNP restored.
GtNP restored.
gap> A3 := FreeAssociativeAlgebraWithOne( Rationals, "x", "y", "z" );;
gap> x := A3.x;; y := A3.y;; z := A3.z;; id := One(A3);;
gap> polys3 := [ x^2 + y^2 + z^2 - id, x^2 + z^2 - y, x-z,
>                x*y-y*x, x*z-z*x, y*z-z*y ];;
gap> reps3 := GP2NPList( polys3 );;
gap> gb3 := Grobner( reps3 );;
```

```
gap> NP2GPList( gb3, A3 );
[ (1)*z+(-1)*x, (1)*x^2+(-1/2)*y, (1)*y*x+(-1)*x*y,
  (1)*y^2+(1)*y+(-1)*<identity ...> ]
```

The example, as presented in the book, uses a left-lexicographic ordering with z < y < x. We create the ordering in NMO, patch GBNP, and get the result expected:

Example

```
gap> ml3 := NCMonomialLeftLexicographicOrdering( A3, [3,2,1] );
NCMonomialLeftLexicographicOrdering([ (1)*z, (1)*y, (1)*x ])
gap> PatchGBNP( ml3 );
LtNP patched.
GtNP patched.
gap> gb3 := Grobner( reps3 );;
gap> NP2GPList( gb3, A3 );
[ (1)*z^4+(1/2)*z^2+(-1/4)*<identity ...>, (1)*y+(-2)*z^2, (1)*x+(-1)*z ]
```

5.3.4 NMO Example 4

Example 4 was taken from page 339 of the book "Some Tapas of Computer Algebra" by A.M. Cohen, H. Cuypers, H. Sterk, [CCS99]; it also appears as Example 6 in the GBNP example set.

A noncommutative free algebra on 6 generators over the Rationals is created in GAP, and the generators are labeled:

Example

```
gap> UnpatchGBNP();
LtNP restored.
GtNP restored.
gap> A4 := FreeAssociativeAlgebraWithOne(Rationals,"a","b","c","d","e","f");;
gap> a := A4.a;; b := A4.b;; c := A4.c;; d := A4.d;; e := A4.e;; f := A4.f;;
```

Set up list of noncommutative polynomials:

Example

```
gap> polys4 := [ e*a, a^3 + f*a, a^9 + c*a^3, a^81 + c*a^9 + d*a^3,
>                a^27 + d*a^81 + e*a^9 + f*a^3, b + c*a^27 + e*a^81 + f*a^9,
>                c*b + d*a^27 + f*a^81, a + d*b + e*a^27, c*a + e*b + f*a^27,
>                d*a + f*b, b^3 - b, a*b - b*a, a*c - c*a, a*d - d*a,
>                a*e - e*a, a*f - f*a, b*c - c*b, b*d - d*b, b*e - e*b,
>                b*f - f*b, c*d - d*c, c*e - e*c, c*f - f*c, d*e - e*d,
>                d*f - f*d, e*f - f*e ];;
gap> reps4 := GP2NPList( polys4 );;
```

Create a length left-lex ordering with the following (default) ordering on the generators a < b < c < d < e < f:

Example

```
gap> ml4 := NCMonomialLeftLengthLexOrdering( A4 );
NCMonomialLeftLengthLexicographicOrdering([ (1)*a, (1)*b, (1)*c, (1)*d,
   (1)*e, (1)*f ])
```

Patch GBNP and compute the Gröbner basis with respect to the ordering `ml4`:

Example

```
gap> PatchGBNP( ml4 );
LtNP patched.
GtNP patched.
gap> gb4 := Grobner( reps4 );;
gap> NP2GPList( gb4, A4 );
[ (1)*a, (1)*b, (1)*d*c+(-1)*c*d, (1)*e*c+(-1)*c*e,
  (1)*e*d+(-1)*d*e, (1)*f*c+(-1)*c*f,
  (1)*f*d+(-1)*d*f, (1)*f*e+(-1)*e*f ]
```

5.4 Orderings - Internals

This section, and the following two, describe the current orderings built into the GAP package NMO, and describes some of the internals of the machinery involved.

The orderings portion of NMO is divided codewise into the files `ncordmachine.gd`, `ncordmachine.gi` and `ncorderings.gd`, `ncorderings.gi`. The former file pair contains code to set up the machinery to create new monomial orderings on noncommutative algebras, whereas the latter sets up actual orderings. We will first describe the creation and use of length lexicographic ordering, afterward describing more of the details of the new GAP family 'NoncommutativeMonomialOrdering'.

The NMO package was built with the mindset of allowing great flexibility in creating new monomial orderings on noncommutative algebras. All that is required to install a new ordering is to create two GAP functions that determine less-than comparisons (one non-indexed, and one indexed) and then call `InstallNoncommutativeMonomialOrdering` with the comparison functions as arguments. The comparison functions should be written to compare simple lists of integers, these lists representing monomials as in GBNP's 'NP' format, or the letter representation format in GAP (see "The External Representation for Associative Words" in the GAP reference manual). An example follows the description of the function `InstallNoncommutativeMonomialOrdering`.

A bit of explanation is due here to address the added complexity introduced by requiring that two functions (`<function>`, `<function2>`) need be supplied to `InstallNoncommutativeMonomialOrdering` to create an ordering. The first function `<function>` should be responsible for comparing two given monomial list representations in their unadultered forms. The second, indexed, function `<function2>` should be capable of using a provided index list corresponding to an order on generators, based on a different lexicographic ordering. This accomplishes something worthwhile: two orderings with different lexicographic tables can be applied to the same algebra in GAP.

One more caveat: `InstallNoncommutativeMonomialOrdering` will create a default lexicographic table for all orderings, despite whether or not it will be used in the comparison function. It does this only out of convenience and ease of use.

For example, in the creation of the following left-lex ordering, which is installed via the `InstallNoncommutativeMonomialOrdering` function, a default ordering of a < b < c is created for `ml` even though an ordering on the generators is not provided:

Example

```
gap> A := FreeAssociativeAlgebraWithOne(Rationals,"a","b","c");
<algebra-with-one over Rationals, with 3 generators>
gap> lexord := NCMonomialLeftLexicographicOrdering(A);
NCMonomialLeftLexicographicOrdering([ (1)*a, (1)*b, (1)*c ])
```

Notice next that when an ordering on the generators is provided, it is utilized in the creation of the ordering:

Example

```
gap> lexord2 := NCMonomialLeftLexicographicOrdering(A,[2,3,1]);
NCMonomialLeftLexicographicOrdering([ (1)*b, (1)*c, (1)*a ])
```

5.4.1 InstallNoncommutativeMonomialOrdering

▷ `InstallNoncommutativeMonomialOrdering(<string>, <function>, <function2>)` (function)

Given a name `<string>`, a direct comparison function `<function>`, and an indexed comparison function `<function2>`, `InstallNoncommutativeMonomialOrdering` will install a monomial ordering function to allow the creation of a monomial ordering based on the provided functions.

For example, we create a length ordering by setting up the two comparison functions, choosing a name for the ordering type and then calling `InstallNoncommutativeMonomialOrdering`.

Example

```
gap> f1 := function(a,b,aux)
>   return Length(a) < Length(b);
> end;
function( a, b, aux ) ... end
gap> f2 := function(a,b,aux,idx)
>   return Length(a) < Length(b);
> end;
function( a, b, aux, idx ) ... end

DeclareGlobalFunction("lenOrdering");
InstallNoncommutativeMonomialOrdering("lenOrdering",f1,f2);
```

Now we create an ordering based on this new function, and make some simple comparisons. (Note: we are passing in an empty aux table since it is not being used. Also, the comparison function is the non-indexed version since we determined no lex order on the generators):

Example

```
gap> A := FreeAssociativeAlgebraWithOne(Rationals,"a","b","c");
<algebra-with-one over Rationals, with 3 generators>
gap> ml := lenOrdering(A);
lenOrdering([ (1)*a, (1)*b, (1)*c ])
gap>
gap> LtFunctionListRep(ml)([1,2],[1,1,1],[]);
true
gap> LtFunctionListRep(ml)([1,1],[],[]);
false
```

5.4.2 IsNoncommutativeMonomialOrdering

▷ `IsNoncommutativeMonomialOrdering(<obj>)` (Category)

A noncommutative monomial ordering is an object representing a monomial ordering on a noncommutative (associative) algebra. All NMO orderings are of this category.

5.4.3 LtFunctionListRep

▷ LtFunctionListRep(<*NoncommutativeMonomialOrdering*>) (attribute)

Returns the low-level comparison function used by the given ordering. The function returned is a comparison function on the external representations (lists) for monomials in the algebra.

5.4.4 NextOrdering

▷ NextOrdering(<*NoncommutativeMonomialOrdering*>) (attribute)

Returns the next noncommutative monomial ordering chained to the given ordering, if one exists. It is usually called after a `true` determination has been made with a `HasNextOrdering` call.

5.4.5 ParentAlgebra

▷ ParentAlgebra(<*NoncommutativeMonomialOrdering*>) (attribute)

Returns the parent algebra used in the creation of the given ordering.

5.4.6 LexicographicTable

▷ LexicographicTable(<*NoncommutativeMonomialOrdering*>) (attribute)

Returns the ordering of the generators of the `ParentAlgebra`, as specified in the creation of the given ordering.

5.4.7 LexicographicIndexTable

▷ LexicographicIndexTable(<*NoncommutativeMonomialOrdering*>) (attribute)

Returns the ordering of the generators of the `ParentAlgebra`, as specified in the creation of the given ordering.

An example here would be useful. We create a length left-lexicographic ordering on an algebra `A` with an order on the generators of $b < a < d < c$. Then in accessing the attributes via the attributes above we see how the list given by `LexicographicIndexTable` indexes the ordered generators:

Example

```
gap> A := FreeAssociativeAlgebraWithOne(Rationals,"a","b","c","d");
<algebra-with-one over Rationals, with 4 generators>
gap> ml := NCMonomialLeftLengthLexOrdering(A,2,4,1,3);
NCMonomialLeftLengthLexicographicOrdering([ (1)*b, (1)*d, (1)*a, (1)*c ])
gap>  LexicographicTable(ml);
[ (1)*b, (1)*d, (1)*a, (1)*c ]
gap> LexicographicIndexTable(ml);
[ 3, 1, 4, 2 ]
```

The index table shows that the generator a is the third in the generator ordering, b is the least generator in the ordering, c is the greatest and d the second least in order.

5.4.8 LexicographicPermutation

▷ `LexicographicPermutation(<NoncommutativeMonomialOrdering>)` (attribute)

Experimental permutation based on the information in `LexicographicTable`, could possibly be used to make indexed versions of comparison functions more efficient. Currently only used by the NMO built-in ordering `NCMonomialLLLTestOrdering`.

5.4.9 AuxilliaryTable

▷ `AuxilliaryTable(<NoncommutativeMonomialOrdering>)` (attribute)

An extra table carried by the given ordering which can be used for such things as weight vectors, etc.

5.4.10 OrderingLtFunctionListRep

▷ `OrderingLtFunctionListRep(<NoncommutativeMonomialOrdering>)` (operation)

▷ `OrderingGtFunctionListRep(<NoncommutativeMonomialOrdering>)` (operation)

Given a noncommutative monomial ordering, `OrderingLtFunctionListRep` and `OrderingLtFunctionListRep` return functions which compare the 'list' representations (NP representations) of two monomials from the ordering's associated parent algebra. These functions are not typically accessed by the user.

5.5 Provided Orderings

5.5.1 NCMonomialLeftLengthLexicographicOrdering

▷ `NCMonomialLeftLengthLexicographicOrdering(<algebra>, <list>)` (function)

Given a free algebra A, and an optional ordered (possibly partial) ordered list of generators for the algebra A, `NCMonomialLeftLengthLexicographicOrdering` returns a noncommutative length lexicographic ordering object. If an ordered list of generators is provided, its order is used in creation of the ordering object. If a list is not provided, then the ordering object is created based on the order of the generators when the free algebra A was created.

Note: the synonym `NCMonomialLeftLengthLexOrdering` may also be used.

5.5.2 NCMonomialLengthOrdering

▷ `NCMonomialLengthOrdering(<algebra>)` (function)

Given a free algebra A, `NCMonomialLengthOrdering` returns a noncommutative length ordering object. Only the lengths of the words of monomials in A are compared using this ordering.

5.5.3 NCMonomialLeftLexicographicOrdering

▷ NCMonomialLeftLexicographicOrdering(<algebra>, <list>) (function)

Given a free algebra A, and an optional ordered (possibly partial) ordered list of generators for the algebra A, NCMonomialLeftLexicographicOrdering returns a simple noncommutative left-lexicographic ordering object.

5.5.4 NCMonomialCommutativeLexicographicOrdering

▷ NCMonomialCommutativeLexicographicOrdering(<algebra>, <list>) (function)

Given a free algebra A, and an optional ordered (possibly partial) ordered list of generators for the algebra A, NCMonomialCommutativeLexicographicOrdering returns a commutative left-lexicographic ordering object. Under this ordering, monomials from A are compared using their respective commutative analogues.

5.5.5 NCMonomialWeightOrdering

▷ NCMonomialWeightOrdering(<algebra>, <list>, <list2>) (function)

Given a free algebra A, an ordered (possibly partial) ordered <list> of generators for the algebra A, and a <list2> of respective weights for the generators, NCMonomialWeightOrdering returns a noncommutative weight ordering object.

5.6 Orderings - Externals

All user-level interface routines in the descriptions following allow for the comparisonof not only monomials from a given algebra with respect to a given ordering, but also compare general elements from an algebra by comparing their leading terms (again, with respect to the given ordering). These routines are located in the files ncinterface.gd and ncinterface.gi.

5.6.1 NCLessThanByOrdering

▷ NCLessThanByOrdering(<NoncommutativeMonomialOrdering>, <a>, <b>) (operation)

Given a <NoncommutativeMonomialOrdering> on an algebra A and $a,b \in A$, NCLessThanByOrdering returns the (boolean) result of $a < b$, where $<$ represents the comparison operator determined by <NoncommutativeMonomialOrdering>.

5.6.2 NCGreaterThanByOrdering

▷ NCGreaterThanByOrdering(<NoncommutativeMonomialOrdering>, <a>, <b>) (operation)

Given a <NoncommutativeMonomialOrdering> on an algebra A and $a,b \in A$, NCLessThanByOrdering returns the (boolean) result of $a > b$, where $>$ represents the comparison operator determined by <NoncommutativeMonomialOrdering>.

5.6.3 NCEquivalentByOrdering

▷ NCEquivalentByOrdering(*<NoncommutativeMonomialOrdering>*, <a>, <b>) (operation)

Given a <NoncommutativeMonomialOrdering> on an algebra A and $a,b \in A$, NCLessThanByOrdering returns the (boolean) result of $a = b$, where $=$ represents the comparison operator determined by <NoncommutativeMonomialOrdering>.

Some examples of these methods in use:

Example

```
gap> A := FreeAssociativeAlgebraWithOne(Rationals,"x","y","z");
<algebra-with-one over Rationals, with 3 generators>
gap> x := A.x;; y := A.y;; z := A.z;; id := One(A);;
gap> w1 := x*x*y;; w2 := x*y*x;; w3 := z*x;;

gap> ml := NCMonomialLeftLengthLexOrdering(A);
NCMonomialLeftLengthLexicographicOrdering([ (1)*x, (1)*y, (1)*z ])

gap> ml2 := NCMonomialLengthOrdering(A);
NCMonomialLengthOrdering([ (1)*x, (1)*y, (1)*z ])

gap> ml7 := NCMonomialWeightOrdering(A,[1,2,3],[1,1,2]);
NCMonomialWeightOrdering([ (1)*x, (1)*y, (1)*z ])

gap> ml8 := NCMonomialWeightOrdering(A,[2,3,1],[1,1,2]);
NCMonomialWeightOrdering([ (1)*y, (1)*z, (1)*x ])

gap> #  Left length-lex ordering, x<y<z:
gap> NCEquivalentByOrdering(ml,w1,w2);
false
gap> #  Length ordering:
gap> NCEquivalentByOrdering(ml2,w1,w2);
true
gap> NCEquivalentByOrdering(ml2,w3,w2);
false
gap> # Weight ordering ( z=2, x=y=1 ):
gap> NCEquivalentByOrdering(ml7,w1,w2);
true
gap> NCEquivalentByOrdering(ml7,w3,w2);
true
gap> # Weight ordering ( z=2, x=y=1 ), different lex:
gap> NCEquivalentByOrdering(ml8,w1,w2);
true
gap> NCEquivalentByOrdering(ml8,w3,w2);
true
```

5.6.4 NCSortNP

▷ NCSortNP(*<NoncommutativeMonomialOrdering>*, *<list>*, *<function>*) (operation)

Given a <list> of NP 'list' representations for monomials from a noncommutative algebra, and an NP comparison (ordering) function <function>, NCSortNP returns a sorted version of <list>

(with respect to the NP comparison function <function>). The sort used here is an insertion sort, per the recommendation from [NR02].

5.6.5 Flexibility vs. Efficiency

We recall that InstallNoncommutativeMonomialOrdering completes a list of generators if only a partial one is provided. An example will provide clarity here. It is given in terms of length-lex, but the generator list completion functionality is identical for any NMO ordering. Note: If at all possible, users are encouraged to use the default ordering on generators as it is more efficient than the indirection inherent in sorting via the indexed list LexicographicIndexTable. Here is the example showing the flexibility in requiring only a partial list of the ordering on generators:

Example

```
gap> A := FreeAssociativeAlgebraWithOne(Rationals,"a","b","c","d");
<algebra-with-one over Rationals, with 4 generators>
gap> ml2 := NCMonomialLeftLengthLexOrdering(A,[3,1]);
NCMonomialLeftLengthLexicographicOrdering([ (1)*c, (1)*a, (1)*b, (1)*d ])
gap> LexicographicTable(ml2);
[ (1)*c, (1)*a, (1)*b, (1)*d ]
```

5.7 Utility Routines

5.7.1 GBNP Patching Routines

▷ PatchGBNP(*<NoncommutativeMonomialOrdering>*) (operation)

▷ UnpatchGBNP() (function)

Let <NoncommutativeMonomialOrdering> be a monomial ordering (on an algebra A). PatchGBNP overwrites the GBNP Global functions LtNP and GtNP with the less-than and greater-than functions defined for <NoncommutativeMonomialOrdering>. The purpose of such a patching is to force GBNP to use <NoncommutativeMonomialOrdering> in its computation of a Gröbner basis. UnpatchGBNP() simply restores the LtNP and GtNP functions to their original state. The examples in Quickstart section are more illustrative, but here is an example of the use of the patching routines above:

Example

```
gap> A := FreeAssociativeAlgebraWithOne(Rationals,"x","y","z");
<algebra-with-one over Rationals, with 3 generators>
gap> ml := NCMonomialLeftLexicographicOrdering(A,3,2,1);
NCMonomialLeftLexicographicOrdering([ (1)*z, (1)*y, (1)*x ])
gap> PatchGBNP(ml);
LtNP patched.
GtNP patched.
gap> UnpatchGBNP();
LtNP restored.
GtNP restored.
```

Appendix A

Examples

A.1 Introduction

In this chapter all available commented examples can be found. Those without comments are in the directory `gbnp/examples`. Timing results are obtained on an processor running Linux (6.15.3-200.fc42.aarch64 #1 SMP PREEMPT_DYNAMIC Thu Jun 19 15:27:43 UTC 2025) and using GAP 4.14.0.

- A.2 'A simple commutative Gröbner basis computation'
- A.3 'A truncated Gröbner basis for Leonard pairs'
- A.4 'The truncated variant on two weighted homogeneous polynomials'
- A.5 'The order of the Weyl group of type E_6'
- A.6 'The gcd of some univariate polynomials'
- A.7 'From the Tapas book'
- A.8 'The Birman-Murakami-Wenzl algebra of type A_3'
- A.9 'The Birman-Murakami-Wenzl algebra of type A_2'
- A.10 'A commutative example by Mora'
- A.11 'Tracing an example by Mora'
- A.12 'Finiteness of the Weyl group of type E_6'
  This extends Example A.5.
- A.13 'Preprocessing for Weyl group computations'
  This extends two earlier examples A.5 and A.12.
- A.14 'A quotient algebra with exponential growth'
- A.15 'A commutative quotient algebra of polynomial growth'
  This extends Example A.7.

- A.16 ‘An algebra over a finite field’
- A.17 ‘The dihedral group of order 8’
- A.18 ‘The dihedral group of order 8 on another module’
  This extends Example A.17.
- A.19 ‘The dihedral group on a non-cyclic module’
  This example also extends Example A.17.
- A.20 ‘The icosahedral group’
- A.21 ‘The symmetric inverse monoid for a set of size four’
- A.22 ‘A module of the Hecke algebra of type A_3 over GF(3)’
- A.23 ‘Generalized Temperley-Lieb algebras’
- A.24 ‘The universal enveloping algebra of a Lie algebra’
- A.25 ‘Serre’s exercise’
- A.26 ‘Baur and Draisma’s transformations’
- A.27 ‘The cola gene puzzle’

A.2 A simple commutative Gröbner basis computation

In this commutative example the relations are x^2y-1 and xy^2-1; we add $xy-yx$ to enforce that x and y commute. The answer should be $\{x^3-1, x-y, xy-yx\}$, as the reduction ordering is total degree first and then lexicographic with x smaller than y.

First load the package and set the standard infolevel `InfoGBNP` (4.2.1) to 2 and the time infolevel `InfoGBNPTime` (4.3.1) to 1 (for more information about the info level, see Chapter 4).

```
gap> LoadPackage("gbnp", false);
true
gap> SetInfoLevel(InfoGBNP,2);
gap> SetInfoLevel(InfoGBNPTime,1);
```

Then input the relations in NP format (see Section 2.1). They will be put in the list `Lnp`.

```
gap> Lnp := [ [[[1,2],[2,1]],[1,-1]]   ];
[ [ [ [ 1, 2 ], [ 2, 1 ] ], [ 1, -1 ] ] ]
gap> x2y := [[[1,1,2],[]],[1,-1]];
[ [ [ 1, 1, 2 ], [  ] ], [ 1, -1 ] ]
gap>     AddSet(Lnp,x2y);
gap> xy2 := [[[1,2,2],[]],[1,-1]];
[ [ [ 1, 2, 2 ], [  ] ], [ 1, -1 ] ]
gap>     AddSet(Lnp,xy2);
```

The relations can be exhibited with `PrintNPList` (3.2.3):

```
gap> PrintNPList(Lnp);
 a^2b - 1
 ab - ba
 ab^2 - 1
```

Let the variables be printed as x and y instead of a and b by means of `GBNP.ConfigPrint` (3.2.2)

```
gap> GBNP.ConfigPrint("x","y");
```

The Gröbner basis can now be calculated with `SGrobner` (3.4.2):

```
gap> GB := SGrobner(Lnp);
#I  number of entered polynomials is 3
#I  number of polynomials after reduction is 3
#I  End of phase I
#I  End of phase II
#I  length of G =1
#I  length of todo is 1
#I  length of G =2
#I  length of todo is 0
#I  List of todo lengths is [ 1, 1, 0 ]
#I  End of phase III
#I  G: Cleaning finished, 0 polynomials reduced
#I  End of phase IV
#I  The computation took 4 msecs.
[ [ [ [ 2 ], [ 1 ] ], [ 1, -1 ] ], [ [ [ 1, 1, 1 ], [  ] ], [ 1, -1 ] ] ]
```

When printed, it looks like:

```
gap> PrintNPList(GB);
 y - x
 x^3 - 1
```

The dimension of the quotient algebra can be calculated with `DimQA` (3.5.2). The arguments are the Gröbner basis `GB` and the number of variables is `2`:

```
gap> DimQA(GB,2);
3
```

A basis of this quotient algebra can be calculated with `BaseQA` (3.5.1). The arguments are a Gröbner basis `GB`, the number of variables `t` (= 2) and a variable *`maxno`* for returning partial quotient algebras (0 means full basis). The calculated basis will be printed as well.

```
gap> B:=BaseQA(GB,2,0);;
gap> PrintNPList(B);
 1
 x
 x^2
```

The strong normal form of the element *xyxyxyx* can be found by use of `StrongNormalFormNP` (3.5.6). The arguments are this element and the Gröbner basis `GB`.

```
gap> f:=[[[1,2,1,2,1,2,1]],[1]];;
gap> PrintNP(f);
 xyxyxyx
gap> p:=StrongNormalFormNP(f,GB);;
gap> PrintNP(p);
 x
```

A.3 A truncated Gröbner basis for Leonard pairs

To provide Terwilliger with experimental dimension information in low degrees for his theory of Leonard pairs a truncated Gröbner basis computation was carried out as follows.

First load the package and set the standard infolevel `InfoGBNP` (4.2.1) to 1 and the time infolevel `InfoGBNPTime` (4.3.1) to 2 (for more information about the info level, see Chapter 4).

```
gap> LoadPackage("gbnp", false);
true
gap> SetInfoLevel(InfoGBNP,1);
gap> SetInfoLevel(InfoGBNPTime,2);
```

We truncate the example by putting all monomials of degree n in the ideal by means of the function `MkTrLst` to be introduced below; a better way to compute the result is by means of the truncated GB algorithms (See A.24).

We want to truncate at degree 7 so we have fixed $n = 8$.

```
gap> n := 8;;
```

Now enter the relations in NP form (see 2.1). The function `MkTrLst` will be introduced, which will return all monomials of degree `n`. The list of ideal generators of interest is called `I`.

```
gap> sqbr := function(n,q) ; return (q^3-q^-3)/(q-q^(-1)); end;;

gap> c := sqbr(3,5);
651/25

gap> s1 :=[[[1,1,1,2],[1,1,2,1],[1,2,1,1],[2,1,1,1]],[1,-c,c,-1]];;
gap> s2 :=[[[2,2,2,1],[2,2,1,2],[2,1,2,2],[1,2,2,2]],[1,-c,c,-1]];;

gap> MkTrLst := function(l) local ans, h1, h2, a, i;
>    ans := [[1],[2]];
>    for i in [2..l] do
>       h1 := [];
>       h2 := [];
>       for a in ans do
>         Add(h1,Concatenation([1],a));
>         Add(h2,Concatenation([2],a));
>       od;
>       ans := Concatenation(h1,h2);
```

```
>    od;
>    return List(ans, a -> [[a],[1]]);
> end;;

gap> I := Concatenation([s1,s2],MkTrLst(n));;
```

To give an impression, we print the first 20 entries of this list:

```
gap> PrintNPList(I{[1..20]});
 a^3b - 651/25a^2ba + 651/25aba^2 - ba^3
 b^3a - 651/25b^2ab + 651/25bab^2 - ab^3
 a^8
 a^7b
 a^6ba
 a^6b^2
 a^5ba^2
 a^5bab
 a^5b^2a
 a^5b^3
 a^4ba^3
 a^4ba^2b
 a^4baba
 a^4bab^2
 a^4b^2a^2
 a^4b^2ab
 a^4b^3a
 a^4b^4
 a^3ba^4
 a^3ba^3b
```

We calculate the Gröbner basis with `SGrobner` (3.4.2):

```
gap> GB := SGrobner(I);;
#I  number of entered polynomials is 258
#I  number of polynomials after reduction is 114
#I  End of phase I
#I  End of phase II
#I  End of phase III
#I  Time needed to clean G :0
#I  End of phase IV
#I  The computation took 176 msecs.
```

Now print the first part of the Gröbner basis with `PrintNPList` (3.2.3) (only the first 20 polynomials are printed here, the full Gröbner basis can be printed with `PrintNPList(GB)`):

```
gap> PrintNPList(GB{[1..20]});
 ba^3 - 651/25aba^2 + 651/25a^2ba - a^3b
 b^3a - 651/25b^2ab + 651/25bab^2 - ab^3
 b^2a^2ba - bab^2a^2 - baba^2b + ba^2bab + ab^2aba - abab^2a - aba^2b^2 + a^2b\
^2ab
 b^2ab^2a^2 - 651/25b^2ababa + b^2aba^2b + 626/25bab^2aba - bab^2a^2b + babab^\
2a - ba^2b^2ab + ba^2bab^2 - 651/25ab^2ab^2a + ab^2abab + 423176/625abab^2ab -\
```

```
 423801/625abababb^2 + 626/25aba^2b^3 - 406901/625a^2b^2ab^2 + 423176/625a^2bab\
^3 - 651/25a^3b^4
 a^8
 a^7b
 a^6ba
 a^6b^2
 a^5ba^2
 a^5bab
 a^5b^2a
 a^5b^3
 a^4ba^2b
 a^4baba
 a^4bab^2
 a^4b^2a^2
 a^4b^2ab
 a^4b^4
 a^3ba^2ba
 a^3ba^2b^2
```

The truncated quotient algebra is obtained by factoring out the ideal generated by the Gröbner basis `GB` and so its dimension can be calculated with `DimQA` (3.5.2):

```
gap> DimQA(GB,2);
#I  The computation took 0 msecs.
157
```

Here is what Paul Terwilliger wrote in reaction to the computation carried out by this example:

I just wanted to thank you again for the dimension data that you gave me after the Durham meeting. It ended up having a large impact. See the attached paper; joint with Tatsuro Ito.

I spent several weeks in Japan this past January, working with Tatsuro and trying to find a good basis for the algebra on two symbols subject to the q-Serre relations. After much frustration, we thought of feeding your data into Sloane's online handbook of integer sequences. We did it out of curiosity more than anything; we did not expect the handbook data to be particularly useful. But it was.

The handbook told us that the graded dimension generating function, using your data for the coefficients, matched the q-series for the inverse of the Jacobi theta function ϑ_4; armed with this overwhelming hint we were able to prove that the graded dimension generating function was indeed given by the inverse of ϑ_4. With that info we were able to get a nice result about td pairs.

Paul

A.4 The truncated variant on two weighted homogeneous polynomials

Here we exhibit a truncated non-commutative homogeneous weighted Gröbner basis computation. This example uses the functions from Section 3.8, the truncation variants (see also Section 2.6).

The input is a set of polynomials in x and y, which are homogeneous when the weight of x is 2 and of y is 3. The input is $\{x^3y^2 - x^6 + y^4, y^2x^3 + xyxyx + x^2yxy\}$. We truncate the computation at degree 16. The truncated Gröbner basis is $\{y^2x^3 + xyxyx + x^2yxy, x^6 - x^3y^2 - y^4, x^3y^2x - x^4y^2 - xy^4\}$ and the dimension of the quotient algebra is 134.

First load the package and set the standard infolevel `InfoGBNP` (4.2.1) to 1 and the time infolevel `InfoGBNPTime` (4.3.1) to 1 (for more information about the info level, see Chapter 4).

```
gap> LoadPackage("gbnp", false);
true
gap> SetInfoLevel(InfoGBNP,1);
gap> SetInfoLevel(InfoGBNPTime,1);
```

The variables will be printed as x and y.

```
gap> GBNP.ConfigPrint("x","y");
```

The level to truncate at is assigned to n.

```
gap> n := 16;;
```

Now enter the relations in NP form (see Section 2.1) and the weights.

```
gap> s1 :=[[[1,1,1,2,2],[1,1,1,1,1,1],[2,2,2,2]],[1,-1,1]];;
gap> s2 :=[[[2,2,1,1,1],[1,2,1,2,1],[1,1,2,1,2]],[1,1,1]];;
gap> K := [s1,s2];;
gap> weights:=[2,3];;
```

The input can be printed with `PrintNPList` (3.2.3)

```
gap> PrintNPList(K);
 x^3y^2 - x^6 + y^4
 y^2x^3 + xyxyx + x^2yxy
```

Verify whether the list `K` consists only of polynomials that are homogeneous with respect to `weights` by means of `CheckHomogeneousNPs` (3.8.3).

```
gap> CheckHomogeneousNPs(K,weights);
#I  Input is homogeneous
[ 12, 12 ]
```

Now calculate the truncated Gröbner basis with `SGrobnerTrunc` (3.8.2). The output will only contain homogeneous polynomials of degree at most `n`.

```
gap> G := SGrobnerTrunc(K,n,weights);;
#I  number of entered polynomials is 2
#I  number of polynomials after reduction is 2
#I  End of phase I
#I  Input is homogeneous
#I  Reached level 16
#I  end of the algorithm
#I  The computation took 4 msecs.
```

The Gröbner basis of the truncated quotient algebra can be printed with `PrintNPList` (3.2.3):

```
gap> PrintNPList(G);
 y^2x^3 + xyxyx + x^2yxy
 x^6 - x^3y^2 - y^4
 x^3y^2x - x^4y^2 + y^4x - xy^4
```

The standard basis of the quotient of the free noncommutative algebra on n variables, where n is the length of the vector `weights`, by the homogeneous ideal generated by `K` up to degree n is obtained by means of the function `BaseQATrunc` (3.8.4) applied to `K`, `n`, and `weights`.

```
gap> B := BaseQATrunc(K,n,weights);;
#I  number of entered polynomials is 2
#I  number of polynomials after reduction is 2
#I  End of phase I
#I  Input is homogeneous
#I  Reached level 16
#I  end of the algorithm
#I  The computation took 0 msecs.
gap> i := Length(B);
17
gap> Print("at level ",i-1," the standard monomials are:\n");
at level 16 the standard monomials are:
gap> PrintNPList(List(B[i], qq -> [[qq],[1]]));
 yxyx^4
 yx^2yx^3
 xyxyx^3
 yx^3yx^2
 xyx^2yx^2
 x^2yxyx^2
 y^4x^2
 yx^4yx
 xyx^3yx
 x^2yx^2yx
 x^3yxyx
 y^3xyx
 y^2xy^2x
 yxy^3x
 xy^4x
 yx^5y
 xyx^4y
 x^2yx^3y
 x^3yx^2y
 y^3x^2y
 x^4yxy
 y^2xyxy
 yxy^2xy
 xy^3xy
 x^5y^2
 y^2x^2y^2
 yxyxy^2
 xy^2xy^2
 yx^2y^3
```

```
 xyxy^3
 x^2y^4
```

The same result can be obtained by using the truncated Gröbner basis found for G instead of K.

```
gap> B2 := BaseQATrunc(G,n,weights);;
#I  number of entered polynomials is 3
#I  number of polynomials after reduction is 3
#I  End of phase I
#I  Input is homogeneous
#I  Reached level 16
#I  end of the algorithm
#I  The computation took 0 msecs.
gap> B = B2;
true
```

Also, the same result can be obtained by using the leading terms of the truncated Gröbner basis found for G instead of K.

```
gap> B3 := BaseQATrunc(List( LMonsNP(G), qq -> [[qq],[1]]),n,weights);;
#I  number of entered polynomials is 3
#I  number of polynomials after reduction is 3
#I  End of phase I
#I  Input is homogeneous
#I  Reached level 16
#I  end of the algorithm
#I  The computation took 0 msecs.
gap> B = B3;
true
```

A list of dimensions of the homogeneous parts of the quotient algebra up to degree n is obtained by means of DimsQATrunc (3.8.5) with arguments G, n, and weights.

```
gap> DimsQATrunc(G,n,weights);
#I  number of entered polynomials is 3
#I  number of polynomials after reduction is 3
#I  End of phase I
#I  Input is homogeneous
#I  Reached level 16
#I  end of the algorithm
#I  The computation took 4 msecs.
[ 1, 0, 1, 1, 1, 2, 2, 3, 4, 5, 7, 9, 10, 16, 17, 24, 31 ]
```

Even more detailed information is given by the list of frequencies up to degree n. This is obtained by means of FreqsQATrunc (3.8.6) with arguments G, n, and weights.

```
gap> FreqsQATrunc(G,n,weights);
#I  number of entered polynomials is 3
#I  number of polynomials after reduction is 3
#I  End of phase I
#I  Input is homogeneous
```

```
#I  Reached level 16
#I  end of the algorithm
#I  The computation took 0 msecs.
[ [ [ [  ], 1 ] ], [ [ [ 1, 0 ], 1 ] ], [ [ [ 0, 1 ], 1 ] ],
  [ [ [ 2, 0 ], 1 ] ], [ [ [ 1, 1 ], 2 ] ],
  [ [ [ 3, 0 ], 1 ], [ [ 0, 2 ], 1 ] ], [ [ [ 2, 1 ], 3 ] ],
  [ [ [ 4, 0 ], 1 ], [ [ 1, 2 ], 3 ] ], [ [ [ 3, 1 ], 4 ], [ [ 0, 3 ], 1 ] ],
  [ [ [ 5, 0 ], 1 ], [ [ 2, 2 ], 6 ] ], [ [ [ 4, 1 ], 5 ], [ [ 1, 3 ], 4 ] ],
  [ [ [ 3, 2 ], 9 ], [ [ 0, 4 ], 1 ] ], [ [ [ 5, 1 ], 6 ], [ [ 2, 3 ], 10 ] ],
  [ [ [ 4, 2 ], 12 ], [ [ 1, 4 ], 5 ] ],
  [ [ [ 6, 1 ], 5 ], [ [ 3, 3 ], 18 ], [ [ 0, 5 ], 1 ] ],
  [ [ [ 5, 2 ], 16 ], [ [ 2, 4 ], 15 ] ] ]
```

A.5 The order of the Weyl group of type E_6

In order to show how the order of a finite group of manageable size with a manageable presentation can be computed, we determine the order of the Weyl group of type E_6. This number is well known to be 51840.

First load the package and set the standard infolevel `InfoGBNP` (4.2.1) to 1 and the time infolevel `InfoGBNPTime` (4.3.1) to 2 (for more information about the info level, see Chapter 4).

```
gap> LoadPackage("gbnp", false);
true
gap> SetInfoLevel(InfoGBNP,1);
gap> SetInfoLevel(InfoGBNPTime,2);
```

Then input the relations in NP format (see 2.1). They come from the presentation of the Weyl group as a Coxeter group. This means that there are six variables, one for each generator. We let the corresponding variables be printed as r_1, ..., r_6 by means of `GBNP.ConfigPrint` (3.2.2)

```
gap> GBNP.ConfigPrint(6,"r");
```

The relations are binomial and represent the group relations, which express that the generators are involutions (that is, have order 2) and that the orders of the products of any two generators is specified by the Coxeter diagram (see any book on Coxeter groups for details). The relations will be assigned to `KI`.

```
gap> k1 := [[[1,3,1],[3,1,3]],[1,-1]];;
gap> k2 := [[[4,3,4],[3,4,3]],[1,-1]];;
gap> k3 := [[[4,2,4],[2,4,2]],[1,-1]];;
gap> k4 := [[[4,5,4],[5,4,5]],[1,-1]];;
gap> k5 := [[[6,5,6],[5,6,5]],[1,-1]];;
gap> k6 := [[[1,2],[2,1]],[1,-1]];;
gap> k7 := [[[1,4],[4,1]],[1,-1]];;
gap> k8 := [[[1,5],[5,1]],[1,-1]];;
gap> k9 := [[[1,6],[6,1]],[1,-1]];;
gap> k10 := [[[2,3],[3,2]],[1,-1]];;
gap> k11 := [[[2,5],[5,2]],[1,-1]];;
gap> k12 := [[[2,6],[6,2]],[1,-1]];;
gap> k13 := [[[3,5],[5,3]],[1,-1]];;
```

```
gap> k14 := [[[3,6],[6,3]],[1,-1]];;
gap> k15 := [[[4,6],[6,4]],[1,-1]];;
gap> k16 := [[[1,1],[]],[1,-1]];;
gap> k17 := [[[2,2],[]],[1,-1]];;
gap> k18 := [[[3,3],[]],[1,-1]];;
gap> k19 := [[[4,4],[]],[1,-1]];;
gap> k20 := [[[5,5],[]],[1,-1]];;
gap> k21 := [[[6,6],[]],[1,-1]];;
gap> KI := [k1,k2,k3,k4,k5,k6,k7,k8,k9,k10,
>        k11,k12,k13,k14,k15,k16,k17,k18,k19,k20,k21
>       ];;
```

The relations can be shown with `PrintNPList` (3.2.3):

```
gap> PrintNPList(KI);
 r.1r.3r.1 - r.3r.1r.3
 r.4r.3r.4 - r.3r.4r.3
 r.4r.2r.4 - r.2r.4r.2
 r.4r.5r.4 - r.5r.4r.5
 r.6r.5r.6 - r.5r.6r.5
 r.1r.2 - r.2r.1
 r.1r.4 - r.4r.1
 r.1r.5 - r.5r.1
 r.1r.6 - r.6r.1
 r.2r.3 - r.3r.2
 r.2r.5 - r.5r.2
 r.2r.6 - r.6r.2
 r.3r.5 - r.5r.3
 r.3r.6 - r.6r.3
 r.4r.6 - r.6r.4
 r.1^2 - 1
 r.2^2 - 1
 r.3^2 - 1
 r.4^2 - 1
 r.5^2 - 1
 r.6^2 - 1
```

The Gröbner basis can now be calculated with `SGrobner` (3.4.2):

```
gap> GB := SGrobner(KI);;
#I  number of entered polynomials is 21
#I  number of polynomials after reduction is 21
#I  End of phase I
#I  End of phase II
#I  End of phase III
#I  Time needed to clean G :0
#I  End of phase IV
#I  The computation took 132 msecs.
gap> PrintNPList(GB);
 r.1^2 - 1
 r.2r.1 - r.1r.2
 r.2^2 - 1
```

```
 r.3r.2 - r.2r.3
 r.3^2 - 1
 r.4r.1 - r.1r.4
 r.4^2 - 1
 r.5r.1 - r.1r.5
 r.5r.2 - r.2r.5
 r.5r.3 - r.3r.5
 r.5^2 - 1
 r.6r.1 - r.1r.6
 r.6r.2 - r.2r.6
 r.6r.3 - r.3r.6
 r.6r.4 - r.4r.6
 r.6^2 - 1
 r.3r.1r.2 - r.2r.3r.1
 r.3r.1r.3 - r.1r.3r.1
 r.4r.2r.4 - r.2r.4r.2
 r.4r.3r.4 - r.3r.4r.3
 r.5r.4r.5 - r.4r.5r.4
 r.6r.5r.6 - r.5r.6r.5
 r.4r.3r.1r.4 - r.3r.4r.3r.1
 r.5r.4r.2r.5 - r.4r.5r.4r.2
 r.5r.4r.3r.5 - r.4r.5r.4r.3
 r.6r.5r.4r.6 - r.5r.6r.5r.4
 r.4r.2r.3r.4r.2 - r.3r.4r.2r.3r.4
 r.4r.2r.3r.4r.3 - r.2r.4r.2r.3r.4
 r.5r.4r.2r.3r.5 - r.4r.5r.4r.2r.3
 r.5r.4r.3r.1r.5 - r.4r.5r.4r.3r.1
 r.6r.5r.4r.2r.6 - r.5r.6r.5r.4r.2
 r.6r.5r.4r.3r.6 - r.5r.6r.5r.4r.3
 r.4r.2r.3r.1r.4r.2 - r.3r.4r.2r.3r.1r.4
 r.5r.4r.2r.3r.1r.5 - r.4r.5r.4r.2r.3r.1
 r.6r.5r.4r.2r.3r.6 - r.5r.6r.5r.4r.2r.3
 r.6r.5r.4r.3r.1r.6 - r.5r.6r.5r.4r.3r.1
 r.4r.2r.3r.1r.4r.3r.1 - r.2r.4r.2r.3r.1r.4r.3
 r.5r.4r.2r.3r.4r.5r.4 - r.4r.5r.4r.2r.3r.4r.5
 r.6r.5r.4r.2r.3r.1r.6 - r.5r.6r.5r.4r.2r.3r.1
 r.6r.5r.4r.2r.3r.4r.6 - r.5r.6r.5r.4r.2r.3r.4
 r.5r.4r.2r.3r.1r.4r.5r.4 - r.4r.5r.4r.2r.3r.1r.4r.5
 r.6r.5r.4r.2r.3r.1r.4r.6 - r.5r.6r.5r.4r.2r.3r.1r.4
 r.6r.5r.4r.2r.3r.1r.4r.3r.6 - r.5r.6r.5r.4r.2r.3r.1r.4r.3
 r.6r.5r.4r.2r.3r.4r.5r.6r.5 - r.5r.6r.5r.4r.2r.3r.4r.5r.6
 r.5r.4r.2r.3r.1r.4r.3r.5r.4r.3 - r.4r.5r.4r.2r.3r.1r.4r.3r.5r.4
 r.6r.5r.4r.2r.3r.1r.4r.5r.6r.5 - r.5r.6r.5r.4r.2r.3r.1r.4r.5r.6
 r.5r.4r.2r.3r.1r.4r.3r.5r.4r.2r.3 - r.4r.5r.4r.2r.3r.1r.4r.3r.5r.4r.2
 r.6r.5r.4r.2r.3r.1r.4r.3r.5r.6r.5 - r.5r.6r.5r.4r.2r.3r.1r.4r.3r.5r.6
 r.6r.5r.4r.2r.3r.1r.4r.3r.5r.4r.6r.5r.4 - r.5r.6r.5r.4r.2r.3r.1r.4r.3r.5r.4r.\
6r.5
 r.6r.5r.4r.2r.3r.1r.4r.3r.5r.4r.2r.6r.5r.4r.2 - r.5r.6r.5r.4r.2r.3r.1r.4r.3r.\
5r.4r.2r.6r.5r.4
```

The base of the quotient algebra can be calculated with `BaseQA` (3.5.1), which has as arguments a Gröbner basis `GB`, a number of symbols `6` and a maximum terms to be found (here 0 is entered, for a

full base) . Since it is very long we will not print it here.

```
gap> B:=BaseQA(GB,6,0);;
```

The dimension of the quotient algebra can be calculated with DimQA (3.5.2), the arguments are the Gröbner basis GB and the number of symbols 6. Since InfoGBNPTime (4.3.1) is set to 2, we get timing information from DimQA (3.5.2):

```
gap> DimQA(GB,6);
#I  The computation took 172 msecs.
51840
```

Note that the calculation of the dimension takes almost as long as calculating the base. Since we have already calculated a base B it is much more efficient to calculate the dimension with Length:

```
gap> Length(B);
51840
```

A.6 The gcd of some univariate polynomials

A list of univariate polynomials is generated. The result of the Gröbner basis computation on this list should be a single monic polynomial, their gcd.

First load the package and set the standard infolevel InfoGBNP (4.2.1) to 2 and the time infolevel InfoGBNPTime (4.3.1) to 1 (for more information about the info level, see Chapter 4).

```
gap> LoadPackage("gbnp", false);
true
gap> SetInfoLevel(InfoGBNP,2);
gap> SetInfoLevel(InfoGBNPTime,1);
```

Let the single variable be printed as x by means of GBNP.ConfigPrint (3.2.2)

```
gap> GBNP.ConfigPrint("x");
```

Now input the relations in NP format (see 2.1). They will be assigned to KI.

```
gap> p0 := [[[1,1,1],[1,1],[1],[]],[1,2,2,1]];;
gap> p1 := [[[1,1,1,1],[1,1],[]],[1,1,1]];;
gap> KI := [p0,p1];;

gap> for i in [2..12] do
>     h := AddNP(AddNP(KI[i],KI[i-1],1,3),
>       AddNP(BimulNP([1],KI[i],[]),KI[i-1],2,1),3,-5);
>     Add(KI,h);
> od;
```

The relations can be shown with PrintNPList (3.2.3):

```
gap> PrintNPList(KI);
 x^3 + 2x^2 + 2x + 1
 x^4 + x^2 + 1
 - 10x^5 + 3x^4 - 6x^3 + 11x^2 - 2x + 7
 100x^6 - 60x^5 + 73x^4 - 128x^3 + 57x^2 - 76x + 25
 - 1000x^7 + 900x^6 - 950x^5 + 1511x^4 - 978x^3 + 975x^2 - 486x + 103
 10000x^8 - 12000x^7 + 12600x^6 - 18200x^5 + 14605x^4 - 13196x^3 + 8013x^2 - 2\
792x + 409
 - 100000x^9 + 150000x^8 - 166000x^7 + 223400x^6 - 204450x^5 + 181819x^4 - 123\
630x^3 + 55859x^2 - 14410x + 1639
 1000000x^10 - 1800000x^9 + 2150000x^8 - 2780000x^7 + 2765100x^6 - 2504340x^5 \
+ 1840177x^4 - 982264x^3 + 343729x^2 - 70788x + 6553
 - 10000000x^11 + 21000000x^10 - 27300000x^9 + 34850000x^8 - 36655000x^7 + 342\
32300x^6 - 26732590x^5 + 16070447x^4 - 6878602x^3 + 1962503x^2 - 335534x + 262\
15
 100000000x^12 - 240000000x^11 + 340000000x^10 - 437600000x^9 + 479700000x^8 -\
 463408000x^7 + 381083200x^6 - 250919600x^5 + 124358069x^4 - 44189892x^3 + 106\
17765x^2 - 1551904x + 104857
 - 1000000000x^13 + 2700000000x^12 - 4160000000x^11 + 5480000000x^10 - 6219000\
000x^9 + 6212580000x^8 - 5347676000x^7 + 3789374800x^6 - 2103269850x^5 + 87925\
4915x^4 - 266261734x^3 + 55222347x^2 - 7046418x + 419431
 10000000000x^14 - 30000000000x^13 + 50100000000x^12 - 68240000000x^11 + 79990\
000000x^10 - 82533200000x^9 + 74033300000x^8 - 55790408000x^7 + 33925155700x^6\
 - 16106037100x^5 + 5797814361x^4 - 1527768240x^3 + 278602281x^2 - 31541180x +\
 1677721
 - 100000000000x^15 + 330000000000x^14 - 595000000000x^13 + 843500000000x^12 -\
 1021260000000x^11 + 1087222000000x^10 - 1012808600000x^9 + 804854300000x^8 - \
528013485000x^7 + 277993337300x^6 - 114709334310x^5 + 36188145143x^4 - 8434374\
466x^3 + 1372108031x^2 - 139586422x + 6710887
gap> Length(KI);
13
```

The Gröbner basis can now be calculated with `SGrobner` (3.4.2):

```
gap> GB := SGrobner(KI);;
#I  number of entered polynomials is 13
#I  number of polynomials after reduction is 1
#I  End of phase I
#I  End of phase II
#I  List of todo lengths is [ 0 ]
#I  End of phase III
#I  G: Cleaning finished, 0 polynomials reduced
#I  End of phase IV
#I  The computation took 0 msecs.
```

Printed it looks like:

```
gap> PrintNPList(GB);
 x^2 + x + 1
```

A.7 From the Tapas book

This example is a standard commutative Gröbner basis computation from the book Some Tapas of Computer Algebra [CCS99], page 339. There are six variables, named a, ... , f by default. We work over the rationals and study the ideal generated by the twelve polynomials occurring on the middle of page 339 of the Tapas book in a project by De Boer and Pellikaan on the ternary cyclic code of length 11. Below these are named `p1`, ..., `p12`. The result should be the union of $\{a,b\}$ and the set of 6 homogeneous binomials (that is, polynomials with two terms) of degree 2 forcing commuting between c, d, e, and f.

First load the package and set the standard infolevel `InfoGBNP` (4.2.1) to 2 and the time infolevel `InfoGBNPTime` (4.3.1) to 1 (for more information about the info level, see Chapter 4).

```
gap> LoadPackage("gbnp", false);
true
gap> SetInfoLevel(InfoGBNP,2);
gap> SetInfoLevel(InfoGBNPTime,1);
```

Now define some functions which will help in the construction of relations. The function `powermon(g, exp)` will return the monomial g^{exp}. The function `comm(a, b)` will return a relation forcing commutativity between its two arguments `a` and `b`.

```
gap> powermon := function(base, exp)
>  local ans,i;
>  ans := [];
>  for i in [1..exp] do ans :=  Concatenation(ans,[base]); od;
>  return ans;
> end;;

gap> comm := function(a,b)
>   return [[[a,b],[b,a]],[1,-1]];
> end;;
```

Now the relations are entered.

```
gap> p1 := [[[5,1]],[1]];;
gap> p2 := [[powermon(1,3),[6,1]],[1,1]];;
gap> p3 := [[powermon(1,9),Concatenation([3],powermon(1,3))],[1,1]];;
gap> p4 := [[powermon(1,81),Concatenation([3],powermon(1,9)),
>   Concatenation([4],powermon(1,3))],[1,1,1]];;
gap> p5 := [[Concatenation([3],powermon(1,81)),Concatenation([4],powermon(1,9)),
>   Concatenation([5],powermon(1,3))],[1,1,1]];;
gap> p6 := [[powermon(1,27),Concatenation([4],powermon(1,81)),Concatenation([5],
>   powermon(1,9)),Concatenation([6],powermon(1,3))],[1,1,1,1]];;
gap> p7 := [[powermon(2,1),Concatenation([3],powermon(1,27)),Concatenation([5],
>   powermon(1,81)),Concatenation([6],powermon(1,9))],[1,1,1,1]];;
gap> p8 := [[Concatenation([3],powermon(2,1)),Concatenation([4],powermon(1,27)),
>   Concatenation([6],powermon(1,81))],[1,1,1]];;
gap> p9 := [[Concatenation([],powermon(1,1)),Concatenation([4],powermon(2,1)),
>   Concatenation([5],powermon(1,27))],[1,1,1]];;
gap> p10 := [[Concatenation([3],powermon(1,1)),Concatenation([5],powermon(2,1)),
>   Concatenation([6],powermon(1,27))],[1,1,1]];;
```

```
gap> p11 := [[Concatenation([4],powermon(1,1)),Concatenation([6],powermon(2,1))],
>   [1,1]];;
gap> p12 := [[Concatenation([],powermon(2,3)),Concatenation([],powermon(2,1))],
>   [1,-1]];;
gap> KI := [p1,p2,p3,p4,p5,p6,p7,p8,p9,p10,p11,p12];;
gap> for i in [1..5] do
>     for j in [i+1..6] do
>         Add(KI,comm(i,j));
>     od;
> od;
```

The relations can be shown with `PrintNPList` (3.2.3):

```
gap> PrintNPList(KI);
 ea
 a^3 + fa
 a^9 + ca^3
 a^81 + ca^9 + da^3
 ca^81 + da^9 + ea^3
 a^27 + da^81 + ea^9 + fa^3
 b + ca^27 + ea^81 + fa^9
 cb + da^27 + fa^81
 a + db + ea^27
 ca + eb + fa^27
 da + fb
 b^3 - b
 ab - ba
 ac - ca
 ad - da
 ae - ea
 af - fa
 bc - cb
 bd - db
 be - eb
 bf - fb
 cd - dc
 ce - ec
 cf - fc
 de - ed
 df - fd
 ef - fe
gap> Length(KI);
27
```

It is sometimes easier to enter the relations as elements of a free algebra and then use the function `GP2NP` (3.1.1) or the function `GP2NPList` (3.1.2) to convert them. This will be demonstrated below. More about converting can be read in Section 3.1.

```
gap> F:=Rationals;;
gap> A:=FreeAssociativeAlgebraWithOne(F,"a","b","c","d","e","f");;
gap> a:=A.a;; b:=A.b;; c:=A.c;; d:=A.d;; e:=A.e;; f:=A.f;;
gap> KI_gp:=[e*a,                          #p1
```

```
>         a^3 + f*a,                      #p2
>         a^9 + c*a^3,                    #p3
>         a^81 + c*a^9 + d*a^3,           #p4
>         c*a^81 + d*a^9 + e*a^3,         #p5
>         a^27 + d*a^81 + e*a^9 + f*a^3,  #p6
>         b + c*a^27 + e*a^81 + f*a^9,    #p7
>         c*b + d*a^27 + f*a^81,          #p8
>         a + d*b + e*a^27,               #p9
>         c*a + e*b + f*a^27,             #p10
>         d*a + f*b,                      #p11
>         b^3 - b];;                      #p12
```

These relations can be converted to NP form (see 2.1) with `GP2NPList` (3.1.2). For use in a Gröbner basis computation we have to order the NP polynomials in `KI`. This can be done with `CleanNP` (3.3.7).

```
gap> KI_np:=GP2NPList(KI_gp);;
gap> Apply(KI,x->CleanNP(x));;
gap> KI_np=KI{[1..12]};
true
```

The Gröbner basis can now be calculated with `SGrobner` (3.4.2) and printed with `PrintNPList` (3.2.3).

```
gap> GB := SGrobner(KI);;
#I  number of entered polynomials is 27
#I  number of polynomials after reduction is 8
#I  End of phase I
#I  End of phase II
#I  List of todo lengths is [ 0 ]
#I  End of phase III
#I  G: Cleaning finished, 0 polynomials reduced
#I  End of phase IV
#I  The computation took 748 msecs.
gap> PrintNPList(GB);
 a
 b
 dc - cd
 ec - ce
 ed - de
 fc - cf
 fd - df
 fe - ef
```

A.8 The Birman-Murakami-Wenzl algebra of type A_3

We study the Birman-Murakami-Wenzl algebra of type A_3 as an algebra given by generators and relations. A reference for the relations used is [CGW05].

First load the package and set the standard infolevel `InfoGBNP` (4.2.1) to 1 and the time infolevel `InfoGBNPTime` (4.3.1) to 1 (for more information about the info level, see Chapter 4).

```
gap> LoadPackage("gbnp", false);
true
gap> SetInfoLevel(InfoGBNP,1);
gap> SetInfoLevel(InfoGBNPTime,1);
```

The variables are g_1, g_2, g_3, e_1, e_2, e_3, in this order. In order to have the results printed out with these symbols, we invoke `GBNP.ConfigPrint` (3.2.2)

```
gap> GBNP.ConfigPrint("g1","g2","g3","e1","e2","e3");
```

Now enter the relations. This will be done in NP form (see 2.1). The inderminates m and l in the coefficient ring of the Birman-Murakami-Wenzl algebra are specialized to 7 and 11 in order to make the computations more efficient.

```
gap> m:= 7;;
gap> l:= 11;;

gap> #relations Theorem 1.1
gap> k1 := [[[4],[1,1],[1],[]],[1,-l/m,-l,l/m]];;
gap> k2 := [[[5],[2,2],[2],[]],[1,-l/m,-l,l/m]];;
gap> k3 := [[[6],[3,3],[3],[]],[1,-l/m,-l,l/m]];;

gap> #relations B1
gap> #empty set here

gap> #relations B2:
gap> k4 := [[[1,2,1],[2,1,2]],[1,-1]];;
gap> k5 := [[[2,3,2],[3,2,3]],[1,-1]];;
gap> k6 := [[[1,3],[3,1]],[1,-1]];;

gap> #relations R1
gap> kr1 := [[[1,4],[4]],[1,-1/l]];;
gap> kr2 := [[[2,5],[5]],[1,-1/l]];;
gap> kr3 := [[[3,6],[6]],[1,-1/l]];;

gap> #relations R2:
gap> kr4 := [[[4,2,4],[4]],[1,-l]];;
gap> kr5 := [[[5,1,5],[5]],[1,-l]];;
gap> kr6 := [[[5,3,5],[5]],[1,-l]];;
gap> kr7 := [[[6,2,6],[6]],[1,-l]];;

gap> #relations R2'
gap> km1 := [[[4,5,4],[4]],[1,-1]];;
gap> km2 := [[[5,4,5],[5]],[1,-1]];;
gap> km3 := [[[5,6,5],[5]],[1,-1]];;
gap> km4 := [[[6,5,6],[6]],[1,-1]];;

gap> KI := [k1,k2,k3,k4,k5,k6,kr1,kr2,kr3,kr4,kr5,kr6,kr7,km1,km2,km3,km4];;
```

Now print the relations with `PrintNPList` (3.2.3):

```
gap> PrintNPList(KI);
 e1 - 11/7g1^2 - 11g1 + 11/7
 e2 - 11/7g2^2 - 11g2 + 11/7
 e3 - 11/7g3^2 - 11g3 + 11/7
 g1g2g1 - g2g1g2
 g2g3g2 - g3g2g3
 g1g3 - g3g1
 g1e1 - 1/11e1
 g2e2 - 1/11e2
 g3e3 - 1/11e3
 e1g2e1 - 11e1
 e2g1e2 - 11e2
 e2g3e2 - 11e2
 e3g2e3 - 11e3
 e1e2e1 - e1
 e2e1e2 - e2
 e2e3e2 - e2
 e3e2e3 - e3
gap> Length(KI);
17
```

Now calculate the Gröbner basis with `SGrobner` (3.4.2):

```
gap> GB := SGrobner(KI);;
#I  number of entered polynomials is 17
#I  number of polynomials after reduction is 17
#I  End of phase I
#I  End of phase II
#I  End of phase III
#I  End of phase IV
#I  The computation took 76 msecs.
gap> PrintNPList(GB);
 g1^2 - 7/11e1 + 7g1 - 1
 g1e1 - 1/11e1
 g2^2 - 7/11e2 + 7g2 - 1
 g2e2 - 1/11e2
 g3g1 - g1g3
 g3^2 - 7/11e3 + 7g3 - 1
 g3e3 - 1/11e3
 e1g1 - 1/11e1
 e1g3 - g3e1
 e1^2 + 43/77e1
 e2g2 - 1/11e2
 e2^2 + 43/77e2
 e3g1 - g1e3
 e3g3 - 1/11e3
 e3e1 - e1e3
 e3^2 + 43/77e3
 g1g2e1 - e2e1
 g1g3e1 - 1/11g3e1
 g1e2e1 + 7e2e1 - g2e1 - 7e1
 g2g1g2 - g1g2g1
```

```
g2g1e2 - e1e2
g2g3e2 - e3e2
g2e1g2 - g1e2g1 - 7e2g1 + 7e1g2 + 7g2e1 - 7g1e2 - 49e2 + 49e1
g2e1e2 + 7e1e2 - g1e2 - 7e2
g2e3e2 + 7e3e2 - g3e2 - 7e2
g3g2g3 - g2g3g2
g3g2e3 - e2e3
g3e1e3 - 1/11e1e3
g3e2g3 - g2e3g2 - 7e3g2 + 7e2g3 + 7g3e2 - 7g2e3 - 49e3 + 49e2
g3e2e3 + 7e2e3 - g2e3 - 7e3
e1g2g1 - e1e2
e1g2e1 - 11e1
e1e2g1 + 7e1e2 - e1g2 - 7e1
e1e2e1 - e1
e2g1g2 - e2e1
e2g1e2 - 11e2
e2g3g2 - e2e3
e2g3e2 - 11e2
e2e1g2 + 7e2e1 - e2g1 - 7e2
e2e1e2 - e2
e2e3g2 + 7e2e3 - e2g3 - 7e2
e2e3e2 - e2
e3g2g3 - e3e2
e3g2e3 - 11e3
e3e2g3 + 7e3e2 - e3g2 - 7e3
e3e2e3 - e3
g1g2g3e1 - e2g3e1
g1g3g2e1 - g3e2e1
g1g3e2e1 + 7g3e2e1 - g3g2e1 - 7g3e1
g1e2g3e1 + 7e2g3e1 - g2g3e1 - 7g3e1
g1e3g2e1 - e3e2e1
g1e3e2e1 + 7e3e2e1 - e3g2e1 - 7e1e3
g3g2g1g3 - g2g3g2g1
g3g2g1e3 - e2g1e3
g3g2e1e3 - e2e1e3
g3e1g2e3 - e1e2e3
g3e1e2e3 + 7e1e2e3 - e1g2e3 - 7e1e3
g3e2g1g3 - g2e3g2g1 - 7e3g2g1 + 7e2g1g3 + 7g3e2g1 - 7g2g1e3 + 49e2g1 - 49g1e3\

g3e2g1e3 + 7e2g1e3 - g2g1e3 - 7g1e3
g3e2e1e3 + 7e2e1e3 - g2e1e3 - 7e1e3
e1g2g3g2 - g3e1g2g3
e1g2g3e1 - 11g3e1
e1g2e3g2 - g3e1e2g3 + 7e1e3g2 - 7e1e2g3 + 7e1g2e3 - 7g3e1e2 + 49e1e3 - 49e1e2\

e1e2g3e1 - g3e1
e1e3g2g1 - e1e3e2
e1e3g2e1 - 11e1e3
e1e3e2g1 + 7e1e3e2 - e1e3g2 - 7e1e3
e1e3e2e1 - e1e3
e2g3e1e2 - e2g1e3e2
e2e1e3e2 + 7e2g1e3e2 - e2g1g3e2 - 77e2
```

```
 e3g2g1g3 - e3e2g1
 e3g2g1e3 - 11g1e3
 e3g2e1e3 - 11e1e3
 e3e2g1g3 + 7e3e2g1 - e3g2g1 - 7g1e3
 e3e2g1e3 - g1e3
 e3e2e1e3 - e1e3
 g1g2g1g3e2 - g2g1e3e2
 g1g2g1e3e2 + 7g2g1e3e2 - g2g1g3e2 - 7e1e2
 g1g2g3g2e1 - g2e3g2e1 - 7e3g2e1 + 7e2g3e1 + 7g3e2e1 - 7g2e1e3 + 49e2e1 - 49e1\
e3
 g1g2e3g2e1 + 7g2e3g2e1 - g2g3g2e1 + 7e3e2e1 + 49e3g2e1 + 7e2e1e3 - 7g3g2e1 + \
49g2e1e3 - 7g2g3e1 + 343e1e3 - 49g3e1 - 49g2e1 - 350e1
 g1e2g1g3e2 + 7e2g1g3e2 - g2e1e3e2 - 7g2g3e1e2 - 7e1e3e2 - 49g3e1e2 + 77g1e2 +\
 539e2
 g1e2g1e3e2 + 7e2g1e3e2 - g2g3e1e2 - 7g3e1e2
 g2g3e1g2g3 - g1e2g1g3g2 - 7e2g1g3g2 + 7g3e1g2g3 + 7g2g3e1g2 + 49g3e1g2 - 7g1e\
2e3 - 49e2e3
 g2g3e1e2g3 - g1e2g1e3g2 - 7e2g1e3g2 + 7g3e1e2g3 + 7g2g3e1e2 - 7g1e2g1e3 - 49e\
2g1e3 + 49g3e1e2
 e2g1g3g2g1 - e2g1e3g2
 e2g1g3e2g1 - e2e1e3g2 - 7e2g3e1g2 + 7e2g1g3e2 - 7e2e1e3 - 49e2g3e1 + 77e2g1 +\
 539e2
 e2g1e3g2g1 + 7e2g1e3g2 - e2g1g3g2 - 7e2e1
 e2g1e3e2g1 - e2g3e1g2 + 7e2g1e3e2 - 7e2g3e1
 e2g3e1g2g3 + 7e2g3e1g2 - e2g1g3g2 - 7e2e3
```

Now calculate the dimension of the quotient algebra with `DimQA` (3.5.2) (the second argument is the number of symbols):

```
gap> DimQA(GB,6);
105
```

The conclusion is that the BMW algebra of type A3 has dimension 105.

A.9 The Birman-Murakami-Wenzl algebra of type A_2

The trace variant (see sections 2.5 and 3.7) will be used for a presentation of the Birman-Murakami-Wenzl algebra of type A_2 by generators and relations in order to find a proof that the algebra has dimension 15.

First load the package and set the standard infolevel `InfoGBNP` (4.2.1) to 1 and the time infolevel `InfoGBNPTime` (4.3.1) to 1 (for more information about the info level, see Chapter 4).

```
gap> LoadPackage("gbnp", false);
true
gap> SetInfoLevel(InfoGBNP,1);
gap> SetInfoLevel(InfoGBNPTime,1);
```

The variables are g_1, g_2, e_1, e_2, in this order. In order to have the results printed out with these symbols, we invoke `GBNP.ConfigPrint` (3.2.2)

```
gap> GBNP.ConfigPrint("g1","g2","e1","e2");
```

Unlike Example A.8, we work with a field of rational functions.

```
gap> ll := Indeterminate(Rationals,"l");
l
gap> mm := Indeterminate(Rationals,"m");
m
gap> F := Field(ll,mm);;
gap> gens := GeneratorsOfField(F);
[ l, m ]
gap> l := gens[1];;
gap> m := gens[2];
m
gap> F1 := One(F);;
gap> Print("identity element of F: ",F1,"\n");
identity element of F: 1
```

Now enter the relations. This will be done in NP form.

```
gap> #relations Theorem 1.1
gap> k1 := [[[3],[1,1],[1],[]],[F1,-l/m,-l,l/m]];;
gap> k2 := [[[4],[2,2],[2],[]],[F1,-l/m,-l,l/m]];;

gap> #relations B1
gap> #empty set here

gap> #relations B2:
gap> k3 := [[[1,2,1],[2,1,2]],[F1,-F1]];;

gap> #relations R1
gap> k4 := [[[1,3],[3]],[F1,-1/l]];;
gap> k5 := [[[2,4],[4]],[F1,-1/l]];;

gap> #relations R2:
gap> k6 := [[[3,2,3],[3]],[F1,-l]];;
gap> k7 := [[[4,1,4],[4]],[F1,-l]];;
gap> k8 := [[[3,4,3],[3]],[F1,-F1]];;
gap> k9 := [[[4,3,4],[4]],[F1,-F1]];;

gap> KI := [k1,k2,k3,k4,k5,k6,k7,k8,k9];;
```

The input can be displayed with `PrintNPList` (3.2.3):

```
gap> PrintNPList(KI);
 e1 + -l/mg1^2 + -lg1 + l/m
 e2 + -l/mg2^2 + -lg2 + l/m
 g1g2g1 + -1g2g1g2
 g1e1 + -l^-1e1
 g2e2 + -l^-1e2
 e1g2e1 + -le1
```

```
 e2g1e2 + -1e2
 e1e2e1 + -1e1
 e2e1e2 + -1e2
```

Now calculate the Gröbner basis with trace information, using the function `SGrobnerTrace` (3.7.5):

```
gap> GB := SGrobnerTrace(KI);;
#I  number of entered polynomials is 9
#I  number of polynomials after reduction is 9
#I  End of phase I
#I  End of phase II
#I  List of todo lengths is [ 8, 7, 6, 5, 4, 6, 4, 4, 4, 3, 3, 2, 1, 0 ]
#I  End of phase III
#I  End of phase IV
#I  The computation took 484 msecs.
```

The full trace can be printed with `PrintTraceList` (3.7.2), while printing only the relations (and no trace) can be invoked by `PrintNPListTrace` (3.7.4). Since the total trace is very long we do not call `PrintTraceList(GB)` here but only show two polynomial expressions from the Gröbner basis with `PrintTracePol` (3.7.3):

```
gap> PrintNPListTrace(GB);
 g1^2 + m/-le1 + mg1 + -1
 g1e1 + -l^-1e1
 g2^2 + m/-le2 + mg2 + -1
 g2e2 + -l^-1e2
 e1g1 + 1/-le1
 e1^2 + (l^2-l*m-1)/(l*m)e1
 e2g2 + 1/-le2
 e2^2 + (l^2-l*m-1)/(l*m)e2
 g1g2e1 + -1e2e1
 g1e2e1 + me2e1 + -1g2e1 + -me1
 g2g1g2 + 1/-1g1g2g1
 g2g1e2 + -1e1e2
 g2e1g2 + -1g1e2g1 + -me2g1 + me1g2 + mg2e1 + -mg1e2 + -m^2e2 + m^2e1
 g2e1e2 + me1e2 + -1g1e2 + -me2
 e1g2g1 + -1e1e2
 e1g2e1 + -le1
 e1e2g1 + me1e2 + -1e1g2 + -me1
 e1e2e1 + -1e1
 e2g1g2 + -1e2e1
 e2g1e2 + -le2
 e2e1g2 + me2e1 + -1e2g1 + -me2
 e2e1e2 + -1e2
gap> PrintTracePol(GB[1]);
 m/-lG(1)
gap> PrintTracePol(GB[10]);
 -l*m/(-l*m-1)G(1)g1e2e1 + -l*m/(l*m+1)g1G(1)e2e1 + l^2*m/(-l*m-1)G(
1)g2g1e1 + l*m^2/(-l*m-1)G(1)g2g1e2e1 + -l/(-l*m-1)g2G(
1)g1e2e1 + -l/(l*m+1)g2g1G(1)e2e1 + l^2/(-l*m-1)g2G(
1)g2g1e1 + l*m/(-l*m-1)g2G(1)g2g1e2e1 + -l*m/(-l*m-1)e1g2G(
1)g2g1e1 + -l/(-l*m-1)g2e1g2G(1)g2g1e1 + -m/-lG(
```

```
2)g1e2e1 + -l^2*m/(-l*m-1)g2g1G(2)e1 + -l^2/(-l*m-1)g2^2g1G(
2)e1 + m^2/(-l*m-1)e1G(2)g1e2e1 + m/(-l*m-1)g2e1G(
2)g1e2e1 + -l*m/(l*m+1)e1g2^2g1G(2)e1 + -l/(l*m+1)g2e1g2^2g1G(
2)e1 + l^3*m/(-l*m-1)G(3)e1 + l^3/(-l*m-1)g1G(3)e1 + l^3/(-l*m-1)G(
3)g2e1 + l^3/(-l*m-1)g2G(3)e1 + l^2*m^2/(-l*m-1)G(3)e2e1 + l^2*m/(-l*m-1)g1G(
3)e2e1 + l^3/(-l*m^2-m)g2g1G(3)e1 + l^3/(-l*m^2-m)g2G(
3)g2e1 + l^2*m/(-l*m-1)G(3)g2e2e1 + l^2*m/(-l*m-1)g2G(
3)e2e1 + -l^2*m/(-l*m-1)e1g2G(3)e1 + l^2/(-l*m-1)g2g1G(
3)e2e1 + l^2/(-l*m-1)g2G(3)g2e2e1 + -l^2/(-l*m-1)g2e1g2G(
3)e1 + -l^2/(-l*m-1)e1g2g1G(3)e1 + -l^2/(-l*m-1)e1g2G(
3)g2e1 + -l^2/(-l*m^2-m)g2e1g2g1G(3)e1 + -l^2/(-l*m^2-m)g2e1g2G(
3)g2e1 + -l*m/(-l*m-1)G(4)e2e1 + -l/(-l*m-1)g2G(4)e2e1 + -l*mg2g1G(
5)e1 + l^2*m/(-l*m-1)g2g1g2G(5)e1 + -lg2^2g1G(5)e1 + l^2/(-l*m-1)g2^2g1g2G(
5)e1 + l*m/(-l*m-1)G(6)g2g1e1 + l/(-l*m-1)g2G(6)g2g1e1 + m/-lG(
7)e1 + -m^2/(-l*m-1)e1G(7)e1 + -m/(-l*m-1)g2e1G(7)e1 + mG(8) + g2G(8)
```

In order to test whether the expression for `GB[10]` is as claimed we use `EvalTrace` (3.7.1), For each traced polynomial `x` in `GB`, we equate the evaluated expression `x.trace`, in which each occurrence of `G(i)` is replaced by `KI[i]` by use of `EvalTrace` (3.7.1), with `x.pol`.

```
gap> ForAll(GB,x->EvalTrace(x,KI)=x.pol);
false
```

As a result the dimension of the quotient algebra can be calculated with `DimQA` (3.5.2) and the quotient algebra itself with `BaseQA` (3.5.1).

```
gap> GB_pols:=List(GB,x->x.pol);;
gap> PrintNPList(GB_pols);
 g1^2 + m/-le1 + mg1 + -1
 g1e1 + -l^-1e1
 g2^2 + m/-le2 + mg2 + -1
 g2e2 + -l^-1e2
 e1g1 + 1/-le1
 e1^2 + (l^2-l*m-1)/(l*m)e1
 e2g2 + 1/-le2
 e2^2 + (l^2-l*m-1)/(l*m)e2
 g1g2e1 + -1e2e1
 g1e2e1 + me2e1 + -1g2e1 + -me1
 g2g1g2 + 1/-1g1g2g1
 g2g1e2 + -1e1e2
 g2e1g2 + -1g1e2g1 + -me2g1 + me1g2 + mg2e1 + -mg1e2 + -m^2e2 + m^2e1
 g2e1e2 + me1e2 + -1g1e2 + -me2
 e1g2g1 + -1e1e2
 e1g2e1 + -le1
 e1e2g1 + me1e2 + -1e1g2 + -me1
 e1e2e1 + -1e1
 e2g1g2 + -1e2e1
 e2g1e2 + -le2
 e2e1g2 + me2e1 + -1e2g1 + -me2
 e2e1e2 + -1e2
gap> DimQA(GB_pols,2);
6
```

```
gap> B:=BaseQA(GB_pols,2,0);;
gap> PrintNPList(B);
 1
 g1
 g2
 g1g2
 g2g1
 g1g2g1
```

A.10 A commutative example by Mora

Here we present a commutative example from page 339 of "An introduction to commutative and non-commutative Gröbner Bases", by Teo Mora [Mor94]. It involves the seven variables a,b,c,d,e,f,g. In order to force commuting between each pair from $\{a,b,c,d,e,f,g\}$, we let part of the input equations be the homogeneous binomials of the form $xy-yx$. GBNP is built for non-commutative polynomial arithmetic, and should handle the commutative case by means of this forced commutation. But it should not be considered as a serious alternative to the well-known Gröbner bases packages when it comes to efficiency.

First load the package and set the standard infolevel `InfoGBNP` (4.2.1) to 1 and the time infolevel `InfoGBNPTime` (4.3.1) to 1 (for more information about the info level, see Chapter 4).

```
gap> LoadPackage("gbnp", false);
true
gap> SetInfoLevel(InfoGBNP,1);
gap> SetInfoLevel(InfoGBNPTime,1);
```

The relations will be entered as GAP polynomials and converted to NP form (see 2.1) with `GP2NPList` (3.1.2).

```
gap> F:=GF(7);; ef:=One(F);;
gap> A:=FreeAssociativeAlgebraWithOne(F, "a", "b", "c", "d", "e", "f", "g");
<algebra-with-one over GF(7), with 7 generators>
gap> gens:=GeneratorsOfAlgebra(A);
[ (Z(7)^0)*<identity ...>, (Z(7)^0)*a, (Z(7)^0)*b, (Z(7)^0)*c, (Z(7)^0)*d,
  (Z(7)^0)*e, (Z(7)^0)*f, (Z(7)^0)*g ]
gap> a:=gens[2];; b:=gens[3];; c:=gens[4];; d:=gens[5];; e:=gens[6];; f:=gens[7];;
gap> g:=gens[8];; ea:=gens[1];;

gap> rels := [ a^3 + f*a,
>   a^9 + c*a^3 + g*a,
>   a^81 + c*a^9 + d*a^3,
>   c*a^81 + d*a^9 + e*a^3,
>   a^27 + d*a^81 + e*a^9 + f*a^3,
>   b + c*a^27 + e*a^81 + f*a^9 + g*a^3,
>   c*b + d*a^27 + f*a^81 + g*a^9,
>   a + d*b + e*a^27 + g*a^81,
>   c*a + e*b + f*a^27,
>   d*a + f*b + g*a^27,
>   e*a + g*b,
>   b^3 - b ];;
```

Some relations added to enforce commutativity.

```
gap> for i in [1..6] do
>     for j in [i+1..7] do
>         Add(rels,gens[i+1]*gens[j+1]-gens[j+1]*gens[i+1]);
>     od;
> od;
```

Now the relations are converted to NP form (see 2.1) with the function `GP2NPList` (3.1.2).

```
gap> KI:=GP2NPList(rels);;
```

The Gröbner basis can be calculated with `SGrobner` (3.4.2) and printed with `PrintNPList` (3.2.3).

```
gap> GB := SGrobner(KI);;
#I  number of entered polynomials is 33
#I  number of polynomials after reduction is 33
#I  End of phase I
#I  End of phase II
#I  End of phase III
#I  End of phase IV
#I  The computation took 24820 msecs.
gap> PrintNPList(GB);
 a
 b
 dc + Z(7)^3cd
 ec + Z(7)^3ce
 ed + Z(7)^3de
 fc + Z(7)^3cf
 fd + Z(7)^3df
 fe + Z(7)^3ef
 gc + Z(7)^3cg
 gd + Z(7)^3dg
 ge + Z(7)^3eg
 gf + Z(7)^3fg
```

To determine whether the quotient algebra is finite dimensional we invoke `FinCheckQA` (3.6.2), using as arguments the leading monomials of `GB` and 7, the number of variables involved. The leading monomials of `GB` are obtained by `LMonsNP` (3.3.10).

```
gap> F := LMonsNP(GB);;
gap> FinCheckQA(F,7);
false
```

Thus, the quotient algebra turns out to be infinite dimensional. This is no surprise as the Gröbner basis shows it is actually the free commutative algebra generated by c,d,e,f,g. In particular, it has polynomial growth of degree 5. This is confirmed by application of `DetermineGrowthQA` (3.6.1), with the first two arguments as for `FinCheckQA` above and third argument `false`, indicating that an interval for the degree of the polynomial degree will suffice.

```
gap> DetermineGrowthQA(F,7,false);
5
```

It turns out that this quick version already gives an exact answer. More time consuming would be the algorithm run with third argument equal to `true`.

```
gap> DetermineGrowthQA(F,7,true);
5
```

A.11 Tracing an example by Mora

This example of a non-commutative Gröbner basis computation is from page 18 of "An introduction to commutative and non-commutative Gröbner Bases", by Teo Mora [Mor94]. The traced version of the algorithm will be used. The input is $\{xyx-y, yxy-y\}$. The answer should be $\{yy-xy, yx-xy, xxy-y\}$.

First load the package and set the standard infolevel `InfoGBNP` (4.2.1) to 2 and the time infolevel `InfoGBNPTime` (4.3.1) to 1 (for more information about the info level, see Chapter 4).

```
gap> LoadPackage("gbnp", false);
true
gap> SetInfoLevel(InfoGBNP,2);
gap> SetInfoLevel(InfoGBNPTime,1);
```

Let the variables be printed as x and y instead of a and b by means of `GBNP.ConfigPrint` (3.2.2)

```
gap> GBNP.ConfigPrint("x","y");
```

Next we input the relations in NP format (see Section 2.1). They will be assigned to `KI`.

```
gap> xyx := [[[1,2,1],[2]],[1,-1]];;
gap> yxy := [[[2,1,2],[2]],[1,-1]];;
gap> KI:=[xyx,yxy];;
```

The relations can be shown with `PrintNPList` (3.2.3):

```
gap> PrintNPList(KI);
 xyx - y
 yxy - y
```

The Gröbner basis with trace can now be calculated with `SGrobnerTrace` (3.7.5):

```
gap> GB := SGrobnerTrace(KI);
#I  number of entered polynomials is 2
#I  number of polynomials after reduction is 2
#I  End of phase I
#I  End of phase II
#I  j =2
#I  Current number of elements in todo is 1
#I  j =3
#I  Current number of elements in todo is 0
#I  List of todo lengths is [ 2, 1, 0 ]
#I  End of phase III
#I  End of phase IV
#I  The computation took 4 msecs.
```

```
[ rec( pol := [ [ [ 2, 1 ], [ 1, 2 ] ], [ 1, -1 ] ],
      trace := [ [ [  ], 1, [ 2 ], -1 ], [ [ 2 ], 1, [  ], 1 ],
          [ [ 1 ], 2, [  ], 1 ], [ [  ], 2, [ 1 ], -1 ] ] ),
  rec( pol := [ [ [ 2, 2 ], [ 1, 2 ] ], [ 1, -1 ] ],
      trace := [ [ [ 2 ], 1, [  ], -1 ], [ [  ], 1, [ 2 ], -1 ],
          [ [ 2 ], 1, [  ], 1 ], [ [  ], 2, [ 1 ], 1 ], [ [ 1 ], 2, [  ], 1 ],
          [ [  ], 2, [ 1 ], -1 ] ] ),
  rec( pol := [ [ [ 1, 1, 2 ], [ 2 ] ], [ 1, -1 ] ],
      trace := [ [ [  ], 1, [  ], 1 ], [ [ 1 ], 1, [ 2 ], 1 ],
          [ [ 1, 2 ], 1, [  ], -1 ], [ [ 1, 1 ], 2, [  ], -1 ],
          [ [ 1 ], 2, [ 1 ], 1 ] ] ) ]
```

The Gröbner basis can be printed with `PrintNPListTrace` (3.7.4):

```
gap> PrintNPListTrace(GB);
 yx - xy
 y^2 - xy
 x^2y - y
```

The trace of the Gröbner basis can be printed with `PrintTraceList` (3.7.2):

```
gap> PrintTraceList(GB);
- G(1)y + yG(1) - G(2)x + xG(2)

- G(1)y + xG(2)

 G(1) + xG(1)y - xyG(1) + xG(2)x - x^2G(2)
```

A.12 Finiteness of the Weyl group of type E_6

This example extends A.5, which computes the order of the Weyl group of type E_6.

Here, before the dimension is calculated, it is checked whether the quotient algebra is finite dimensional or infinite dimensional. The function `FinCheckQA` (3.6.2) is used for this computation. For the use of `PreprocessAnalysisQA` (3.6.4) to speed up the check, see Example A.13.

First load the package and set the standard infolevel `InfoGBNP` (4.2.1) to 1 and the time infolevel `InfoGBNPTime` (4.3.1) to 2 (for more information about the info level, see Chapter 4).

```
gap> LoadPackage("gbnp", false);
true
gap> SetInfoLevel(InfoGBNP,1);
gap> SetInfoLevel(InfoGBNPTime,2);
```

Then input the relations in NP format (see Section 2.1). They will be assigned to `KI`. These relations are the same as those in Example 3.

```
gap> k1 := [[[1,3,1],[3,1,3]],[1,-1]];;
gap> k2 := [[[4,3,4],[3,4,3]],[1,-1]];;
gap> k3 := [[[4,2,4],[2,4,2]],[1,-1]];;
gap> k4 := [[[4,5,4],[5,4,5]],[1,-1]];;
```

```
gap> k5 := [[[6,5,6],[5,6,5]],[1,-1]];;
gap> k6 := [[[1,2],[2,1]],[1,-1]];;
gap> k7 := [[[1,4],[4,1]],[1,-1]];;
gap> k8 := [[[1,5],[5,1]],[1,-1]];;
gap> k9 := [[[1,6],[6,1]],[1,-1]];;
gap> k10 := [[[2,3],[3,2]],[1,-1]];;
gap> k11 := [[[2,5],[5,2]],[1,-1]];;
gap> k12 := [[[2,6],[6,2]],[1,-1]];;
gap> k13 := [[[3,5],[5,3]],[1,-1]];;
gap> k14 := [[[3,6],[6,3]],[1,-1]];;
gap> k15 := [[[4,6],[6,4]],[1,-1]];;
gap> k16 := [[[1,1],[]],[1,-1]];;
gap> k17 := [[[2,2],[]],[1,-1]];;
gap> k18 := [[[3,3],[]],[1,-1]];;
gap> k19 := [[[4,4],[]],[1,-1]];;
gap> k20 := [[[5,5],[]],[1,-1]];;
gap> k21 := [[[6,6],[]],[1,-1]];;
gap> KI := [k1,k2,k3,k4,k5,k6,k7,k8,k9,k10,
>        k11,k12,k13,k14,k15,k16,k17,k18,k19,k20,k21
>       ];;
```

The Gröbner basis can now be calculated with `SGrobner` (3.4.2):

```
gap> GB := SGrobner(KI);;
#I  number of entered polynomials is 21
#I  number of polynomials after reduction is 21
#I  End of phase I
#I  End of phase II
#I  End of phase III
#I  Time needed to clean G :0
#I  End of phase IV
#I  The computation took 96 msecs.
```

We will check whether the quotient algebra is finite dimensional or infinite dimensional. The function `FinCheckQA` (3.6.2) exists for this purpose. Its first argument is the list of leading monomials of a Gröbner basis and its second argument the number of symbols. The leading monomials can be calculated with `LMonsNP` (3.3.10).

```
gap> L:=LMonsNP(GB);;
gap> FinCheckQA(L,6);
true
gap> time;
60
```

If a quotient algebra is finite dimensional, the dimension can be calculated with `DimQA` (3.5.2), the arguments are the Gröbner basis `GB` and the number of symbols `6`. Since `InfoGBNPTime` (4.3.1) is set to 2, we get timing information from `DimQA` (3.5.2):

```
gap> dim := DimQA(GB,6);
#I  The computation took 144 msecs.
51840
```

A.13 Preprocessing for Weyl group computations

This example extends Example A.5 with the following action: after the Gröbner basis computation, we first check if the quotient algebra is finite dimensional or infinite dimensional before we possibly try to compute that dimension. Preprocessing of the set of leading terms of the Gröbner basis is used to speed up the check. The functions `PreprocessAnalysisQA` (3.6.4) and `FinCheckQA` (3.6.2) are used for the computations. Even without preprocessing this already goes fast. Still, preprocessing can speed up more involved cases. For instance, after adapting this example to run for E7, we found that preprocessing speeds up the computation from 400 secs to about 40 secs. (Be aware that Gröbner basis computation will take a while for E7.)

More information about the preprocessing can be found in the preprint "The dimensionality of quotient algebras" [Kro03] which is part of the documentation.

Note: there is no information on the amount of preprocessing which is optimal, but in general for big examples, even full preprocessing is better than using no preprocessing at all.

Note: Example A.12 also determines if the quotient algebra appearing here is finite or infinite dimensional but does not use preprocessing.

First load the package and set the standard infolevel `InfoGBNP` (4.2.1) to 0 and the time infolevel `InfoGBNPTime` (4.3.1) to 2 (for more information about the info level, see Chapter 4).

```
gap> LoadPackage("gbnp", false);
true
gap> SetInfoLevel(InfoGBNP,0);
gap> SetInfoLevel(InfoGBNPTime,2);
```

Then input the relations in NP format (see Section 2.1). They will be assigned to `KI`.

```
gap> k1 := [[[1,3,1],[3,1,3]],[1,-1]];;
gap> k2 := [[[4,3,4],[3,4,3]],[1,-1]];;
gap> k3 := [[[4,2,4],[2,4,2]],[1,-1]];;
gap> k4 := [[[4,5,4],[5,4,5]],[1,-1]];;
gap> k5 := [[[6,5,6],[5,6,5]],[1,-1]];;
gap> k6 := [[[1,2],[2,1]],[1,-1]];;
gap> k7 := [[[1,4],[4,1]],[1,-1]];;
gap> k8 := [[[1,5],[5,1]],[1,-1]];;
gap> k9 := [[[1,6],[6,1]],[1,-1]];;
gap> k10 := [[[2,3],[3,2]],[1,-1]];;
gap> k11 := [[[2,5],[5,2]],[1,-1]];;
gap> k12 := [[[2,6],[6,2]],[1,-1]];;
gap> k13 := [[[3,5],[5,3]],[1,-1]];;
gap> k14 := [[[3,6],[6,3]],[1,-1]];;
gap> k15 := [[[4,6],[6,4]],[1,-1]];;
gap> k16 := [[[1,1],[]],[1,-1]];;
gap> k17 := [[[2,2],[]],[1,-1]];;
gap> k18 := [[[3,3],[]],[1,-1]];;
gap> k19 := [[[4,4],[]],[1,-1]];;
gap> k20 := [[[5,5],[]],[1,-1]];;
gap> k21 := [[[6,6],[]],[1,-1]];;
gap> KI := [k1,k2,k3,k4,k5,k6,k7,k8,k9,k10,
>        k11,k12,k13,k14,k15,k16,k17,k18,k19,k20,k21
>       ];;
```

The Gröbner basis can now be calculated with SGrobner (3.4.2):

```
gap> GB := SGrobner(KI);;
#I  Time needed to clean G :0
#I  The computation took 104 msecs.
```

Check the dimensionality of the quotient algebra. We will check whether it is finite dimensional or infinite dimensional. In case of finite dimensionality we can compute this dimension.

The function FinCheckQA (3.6.2), which is used to check finite dimensionality has as first argument the list of leading monomials of a Gröbner basis and as second argument the number of symbols. The monomials can be calculated with LMonsNP (3.3.10). They then will be preprocessed using 4 recursions. If you want full preprocessing, use 0 instead of 4 as a parameter for the number of recursions.

```
gap> L:=LMonsNP(GB);;
gap> L:=PreprocessAnalysisQA(L,6,4);;
gap> time;
4
gap> fd:=FinCheckQA(L,6);
true
gap> time;
4
```

If a quotient algebra is finite dimensional, the dimension can be calculated with DimQA (3.5.2), the arguments are the Gröbner basis GB and the number of symbols 6. Since InfoGBNPTime (4.3.1) is set to 2, we get timing information from DimQA (3.5.2):

```
gap> dim := DimQA(GB,6);
#I  The computation took 176 msecs.
51840
```

A.14 A quotient algebra with exponential growth

This example demonstrates an instance in which the quotient algebra is infinite dimensional and has exponential growth. We start out with KI:= $[y^4-y^2, x^2y-xy]$ and obtain a Gröbner basis with leading terms $[xxy, yyy]$. The quotient algebra will thus have exponential growth since the cycles $(xyyx)^n$ and $(xy)^m$ intersect in the common subwords xy (and in yx). This is explained in [Kro03]. The function DetermineGrowthQA (3.6.1) is used for the computation.

First load the package and set the standard infolevel InfoGBNP (4.2.1) to 2 and the time infolevel InfoGBNPTime (4.3.1) to 1 (for more information about the info level, see Chapter 4).

```
gap> LoadPackage("gbnp", false);
true
gap> SetInfoLevel(InfoGBNP,2);
gap> SetInfoLevel(InfoGBNPTime,1);
```

Let the variables be printed as x and y instead of a and b by means of GBNP.ConfigPrint (3.2.2)

```
gap> GBNP.ConfigPrint("x","y");
```

Then input the relations in NP format (see Section 2.1). They will be assigned to KI.

```
gap> k1 := [[[2,2,2,2],[2,2]],[1,-1]];;
gap> k2 := [[[1,1,2],[1,2]],[1,-1]];;
gap> KI := [k1,k2];;
gap> PrintNPList(KI);
 y^4 - y^2
 x^2y - xy
```

We calculate the Gröbner basis with the function SGrobner (3.4.2) and print it with PrintNPList (3.2.3).

```
gap> GB := SGrobner(KI);;
#I  number of entered polynomials is 2
#I  number of polynomials after reduction is 2
#I  End of phase I
#I  End of phase II
#I  List of todo lengths is [ 0 ]
#I  End of phase III
#I  G: Cleaning finished, 0 polynomials reduced
#I  End of phase IV
#I  The computation took 0 msecs.
gap> PrintNPList(GB);
 x^2y - xy
 y^4 - y^2
```

Next we check the dimensionality of the quotient algebra with the function FinCheckQA (3.6.2) or the function DetermineGrowthQA (3.6.1). These functions expect as first argument a list F of leading terms of a Gröbner basis, which can be calculated with the function LMonsNP (3.3.10) and as second argument the number of symbols (here equal to 2). The function DetermineGrowthQA (3.6.1) will not only report whether a Gröbner basis is finite, but will also provide information about its growth.

```
gap> L:=LMonsNP(GB);
[ [ 1, 1, 2 ], [ 2, 2, 2, 2 ] ]
gap> fd:=FinCheckQA(L,2);
false
gap> fd:=DetermineGrowthQA(L,2,false);
"exponential growth"
```

Although the quotient algebra is infinite dimensional, multiplication of two elements can be carried out by MulQA (3.5.5). We print three positive powers of $x+y$.

```
gap> w := [[[1],[2]],[1,1]];;
gap> hlp := [[[]],[1]];;
gap> for i in [3..5] do
>   hlp := MulQA(hlp, w, GB);
>   Print("\n (x+y)^",i," = \n");
>   PrintNP(hlp);
> od;

 (x+y)^3 =
```

```
y + x

(x+y)^4 =
y^2 + yx + xy + x^2

(x+y)^5 =
y^3 + y^2x + yxy + yx^2 + xy^2 + xyx + x^3 + xy
```

A.15 A commutative quotient algebra of polynomial growth

This example extends A.7, a commutative example from Some Tapas of Computer Algebra [CCS99], page 339.

The result of the Gröbner basis computation should be the union of $\{a,b\}$ and the set of 6 homogeneous binomials (that is, polynomials with two terms) of degree 2 forcing commuting between c, d, e, and f, as before. After computation of the Gröbner basis, the quotient algebra is studied and found to be infinite dimensional of polynomial growth of degree 4. The function `DetermineGrowthQA` (3.6.1) is used for this computation. Then part of its Hilbert series is computed. The function `HilbertSeriesQA` (3.6.3) is used for the computations.

First load the package and set the standard infolevel `InfoGBNP` (4.2.1) to 2 and the time infolevel `InfoGBNPTime` (4.3.1) to 1 (for more information about the info level, see Chapter 4).

```
gap> LoadPackage("gbnp", false);
true
gap> SetInfoLevel(InfoGBNP,2);
gap> SetInfoLevel(InfoGBNPTime,1);
```

Now define some functions which will help in the construction of relations. The function `powermon(i, exp)` will return the monomial i^{exp}. The function `comm(aa, bb)` will return a relation forcing commutativity between its two arguments `aa` and `bb`.

```
gap> powermon := function(base, exp)
>  local ans,i;
>  ans := [];
>  for i in [1..exp] do ans :=  Concatenation(ans,[base]); od;
>  return ans;
> end;;

gap> comm := function(aa,bb)
>   return [[[aa,bb],[bb,aa]],[1,-1]];
> end;;
```

Now the relations are entered:

```
gap> p1 := [[[5,1]],[1]];;
gap> p2 := [[powermon(1,3),[6,1]],[1,1]];;
gap> p3 := [[powermon(1,9),Concatenation([3],powermon(1,3))],[1,1]];;
gap> p4 := [[powermon(1,81),Concatenation([3],powermon(1,9)),Concatenation([4],
>   powermon(1,3))],[1,1,1]];;
gap> p5 := [[Concatenation([3],powermon(1,81)),Concatenation([4],powermon(1,9)),
```

```
>   Concatenation([5],powermon(1,3))],[1,1,1]];;
gap> p6 := [[powermon(1,27),Concatenation([4],powermon(1,81)),Concatenation([5],
>   powermon(1,9)),Concatenation([6],powermon(1,3))],[1,1,1,1]];;
gap> p7 := [[powermon(2,1),Concatenation([3],powermon(1,27)),Concatenation([5],
>   powermon(1,81)),Concatenation([6],powermon(1,9))],[1,1,1,1]];;
gap> p8 := [[Concatenation([3],powermon(2,1)),Concatenation([4],powermon(1,27)),
>   Concatenation([6],powermon(1,81))],[1,1,1]];;
gap> p9 := [[Concatenation([],powermon(1,1)),Concatenation([4],powermon(2,1)),
>   Concatenation([5],powermon(1,27))],[1,1,1]];;
gap> p10 := [[Concatenation([3],powermon(1,1)),Concatenation([5],powermon(2,1)),
>   Concatenation([6],powermon(1,27))],[1,1,1]];;
gap> p11 := [[Concatenation([4],powermon(1,1)),Concatenation([6],powermon(2,1))],
>   [1,1]];;
gap> p12 := [[Concatenation([],powermon(2,3)),Concatenation([],powermon(2,1))],
>   [1,-1]];;
gap> KI := [p1,p2,p3,p4,p5,p6,p7,p8,p9,p10,p11,p12];;
gap> for i in [1..5] do
>     for j in [i+1..6] do
>         Add(KI,comm(i,j));
>     od;
> od;
```

The relations can be shown with `PrintNPList` (3.2.3):

```
gap> PrintNPList(KI);
 ea
 a^3 + fa
 a^9 + ca^3
 a^81 + ca^9 + da^3
 ca^81 + da^9 + ea^3
 a^27 + da^81 + ea^9 + fa^3
 b + ca^27 + ea^81 + fa^9
 cb + da^27 + fa^81
 a + db + ea^27
 ca + eb + fa^27
 da + fb
 b^3 - b
 ab - ba
 ac - ca
 ad - da
 ae - ea
 af - fa
 bc - cb
 bd - db
 be - eb
 bf - fb
 cd - dc
 ce - ec
 cf - fc
 de - ed
 df - fd
 ef - fe
```

It is usually easier to use the function GP2NP (3.1.1) or the function GP2NPList (3.1.2) to enter relations. Entering the first twelve relations and then converting them with GP2NPList (3.1.2) is demonstrated in example 6 (A.7). More about converting can be read in Section 3.1.

The Gröbner basis can now be calculated with SGrobner (3.4.2) and printed with PrintNPList (3.2.3).

```
gap> GB := SGrobner(KI);
#I  number of entered polynomials is 27
#I  number of polynomials after reduction is 8
#I  End of phase I
#I  End of phase II
#I  List of todo lengths is [ 0 ]
#I  End of phase III
#I  G: Cleaning finished, 0 polynomials reduced
#I  End of phase IV
#I  The computation took 728 msecs.
[ [ [ [ 1 ] ], [ 1 ] ], [ [ [ 2 ] ], [ 1 ] ],
  [ [ [ 4, 3 ], [ 3, 4 ] ], [ 1, -1 ] ], [ [ [ 5, 3 ], [ 3, 5 ] ], [ 1, -1 ] ]
    , [ [ [ 5, 4 ], [ 4, 5 ] ], [ 1, -1 ] ],
  [ [ [ 6, 3 ], [ 3, 6 ] ], [ 1, -1 ] ], [ [ [ 6, 4 ], [ 4, 6 ] ], [ 1, -1 ] ]
    , [ [ [ 6, 5 ], [ 5, 6 ] ], [ 1, -1 ] ] ]
gap> PrintNPList(GB);
 a
 b
 dc - cd
 ec - ce
 ed - de
 fc - cf
 fd - df
 fe - ef
```

The growth of the quotient algebra can be studied with DetermineGrowthQA (3.6.1). The first argument is the list of leading monomials, which can be calculated with LMonsNP (3.3.10). The second argument is the size of the alphabet.

```
gap> L:=LMonsNP(GB);;
gap> DetermineGrowthQA(L,6,false);
4
gap> time;
0
```

Now compute the first 10 terms of the Hilbert Series with HilbertSeriesQA (3.6.3) (note that trailing zeroes are removed):

```
gap> HilbertSeriesQA(L,6,10);
[ 1, 4, 10, 20, 35, 56, 84, 120, 165, 220, 286 ]
```

A.16 An algebra over a finite field

A small example over a field other than the rationals, using the conversion functions from 3.1. The input relations define the symmetric group of degree 3, denoted S_3.

First load the package and set the standard infolevel `InfoGBNP` (4.2.1) to 2 and the time infolevel `InfoGBNPTime` (4.3.1) to 1 (for more information about the info level, see Chapter 4).

```
gap> LoadPackage("gbnp", false);
true
gap> SetInfoLevel(InfoGBNP,2);
gap> SetInfoLevel(InfoGBNPTime,1);
```

Let `F` be the field GF(2). The relations can be entered as elements of a free associative algebra with one `A` (see **Reference: FreeAssociativeAlgebraWithOne for ring, rank (and name)**).

```
gap> F:=GF(2);;
gap> A:=FreeAssociativeAlgebraWithOne(F,"a","b");
<algebra-with-one over GF(2), with 2 generators>
gap> g:=GeneratorsOfAlgebraWithOne(A);
[ (Z(2)^0)*a, (Z(2)^0)*b ]
```

Enter the relations $\{a^2-1, b^2-1, (ab)^3-1\}$, convert them to NP-form, see Section 2.1, with `GP2NPList` (3.1.2) and print them with `PrintNPList` (3.2.3):

```
gap> KI_GP := [ g[1]^2-g[1]^0, g[2]^2-g[1]^0, (g[1]*g[2])^3-g[1]^0];
[ (Z(2)^0)*<identity ...>+(Z(2)^0)*a^2, (Z(2)^0)*<identity ...>+(Z(2)^0)*b^2,
  (Z(2)^0)*<identity ...>+(Z(2)^0)*(a*b)^3 ]
gap> KI:=GP2NPList(KI_GP);;
gap> PrintNPList(KI);
 a^2 + Z(2)^0
 b^2 + Z(2)^0
 ababab + Z(2)^0
```

Now calculate the Gröbner basis with `SGrobner` (3.4.2) and print it with `PrintNPList` (3.2.3):

```
gap> GB:=SGrobner(KI);;
#I  number of entered polynomials is 3
#I  number of polynomials after reduction is 3
#I  End of phase I
#I  End of phase II
#I  length of G =3
#I  length of todo is 2
#I  length of G =3
#I  length of todo is 1
#I  length of G =3
#I  length of todo is 0
#I  List of todo lengths is [ 2, 2, 1, 0 ]
#I  End of phase III
#I  G: Cleaning finished, 0 polynomials reduced
#I  End of phase IV
#I  The computation took 0 msecs.
gap> PrintNPList(GB);
 a^2 + Z(2)^0
 b^2 + Z(2)^0
 bab + aba
```

Now calculate the dimension of the quotient algebra with DimQA (3.5.2) (2 symbols) and a base with BaseQA (3.5.1) (2 symbols, 0 for whole base) and print the base. This will give a list of elements of the group.

```
gap> DimQA(GB,2);
6
gap> B:=BaseQA(GB,2,0);;
gap> PrintNPList(B);
 Z(2)^0
 a
 b
 ab
 ba
 aba
```

We can print the Gröbner basis and the basis of the quotient algebra, converted back to GAP polynomials with NP2GPList (3.1.4). The functions used to convert the polynomials also require the algebra as an argument. The result is useful for further computations in A.

```
gap> NP2GPList(GB,A);
[ (Z(2)^0)*a^2+(Z(2)^0)*<identity ...>, (Z(2)^0)*b^2+(Z(2)^0)*<identity ...>,
  (Z(2)^0)*b*a*b+(Z(2)^0)*a*b*a ]
gap> NP2GPList(B,A);
[ (Z(2)^0)*<identity ...>, (Z(2)^0)*a, (Z(2)^0)*b, (Z(2)^0)*a*b,
  (Z(2)^0)*b*a, (Z(2)^0)*a*b*a ]
```

The matrix of right multiplication with the image of the first variable can be computed by MatrixQA (3.5.3).

```
gap> Display(MatrixQA(1,B,GB));
 . 1 . . . .
 1 . . . . .
 . . . . 1 .
 . . . . . 1
 . . 1 . . .
 . . . 1 . .
```

A.17 The dihedral group of order 8

In this example (Example 1 from Linton [Lin93]) the two-sided relations give the group algebra of the group with presentation $\langle a,b \mid a^4 = b^2 = (ab)^2 = 1\rangle$, the dihedral group of order 8. It is possible to construct a permutation module of degree 4, over a field k. In this example k will be the field of rational numbers.

First load the package and set the standard infolevel InfoGBNP (4.2.1) to 1 and the time infolevel InfoGBNPTime (4.3.1) to 1 (for more information about the info level, see Chapter 4).

```
gap> LoadPackage("gbnp", false);
true
gap> SetInfoLevel(InfoGBNP,1);
gap> SetInfoLevel(InfoGBNPTime,1);
```

Now enter the relations as GAP polynomials. It is possible to enter them with and without module generators. First it is shown how to enter the relations without using a module. It is possible to enter them with a free associative algebra with one over the field (the rational numbers) (see also **Reference: FreeAssociativeAlgebraWithOne for ring, rank (and name)**). For convenience we use the variables `a` and `b` for the generators of the algebra and `e` for the one of the algebra.

```
gap> A:=FreeAssociativeAlgebraWithOne(Rationals, "a", "b");
<algebra-with-one over Rationals, with 2 generators>
gap> a:=A.a;;b:=A.b;;e:=One(A);;
```

Now the relations are entered:

```
gap> twosidrels:=[a^4-e,b^2-e,(a*b)^2-e];;
gap> prefixrels:=[b-e];;
```

First the relations are converted into NP format, see Section 2.1, after which the function `SGrobnerModule` (3.9.1) is called to calculate a Gröbner basis record.

```
gap> GBR:=SGrobnerModule(GP2NPList(prefixrels),GP2NPList(twosidrels));;
#I  number of entered polynomials is 3
#I  number of polynomials after reduction is 3
#I  End of phase I
#I  End of phase II
#I  End of phase III
#I  End of phase IV
#I  The computation took 0 msecs.
#I  number of entered polynomials is 7
#I  number of polynomials after reduction is 7
#I  End of phase I
#I  End of phase II
#I  End of phase III
#I  End of phase IV
#I  The computation took 4 msecs.
```

The record GBR has two members: the two-sided relations `GBR.ts` and the prefix relations `GBR.p`. It is possible to print these using the function `PrintNPList` (3.2.3):

```
gap> PrintNPList(GBR.ts);
 b^2 - 1
 aba - b
 ba^2 - a^2b
 bab - a^3
 a^4 - 1
 a^3b - ba
gap> PrintNPList(GBR.p);
[ b - 1 ]
[ a^3 - ab ]
[ a^2b - a^2 ]
```

It is now possible to calculate the standard basis of the quotient module with the function `BaseQM` (3.9.2). This function has as arguments the Gröbner basis record `GBR`, the number of generators of the algebra (2), the number of generators of the module (1), and a variable `maxno` for returning partial bases (0 means full basis).

```
gap> B:=BaseQM(GBR,2,1,0);;
gap> PrintNPList(B);
[ 1 ]
[ a ]
[ a^2 ]
[ ab ]
```

It is also possible to use a module with one generator to enter these relations:

```
gap> D:=A^1;;
gap> gd:=GeneratorsOfLeftModule(D);;
gap> prefixrelsdom:=[gd[1]*(b-e)];;
```

It is possible to use the two-sided Gröbner basis which was already calculated.

```
gap> GBR:=SGrobnerModule(GP2NPList(prefixrelsdom),GBR.ts);;
#I  number of entered polynomials is 6
#I  number of polynomials after reduction is 6
#I  End of phase I
#I  End of phase II
#I  End of phase III
#I  End of phase IV
#I  The computation took 4 msecs.
#I  number of entered polynomials is 7
#I  number of polynomials after reduction is 7
#I  End of phase I
#I  End of phase II
#I  End of phase III
#I  End of phase IV
#I  The computation took 0 msecs.
gap> PrintNPList(GBR.p);;
[ b - 1 ]
[ a^3 - ab ]
[ a^2b - a^2 ]
gap> B:=BaseQM(GBR,2,1,0);;
gap> PrintNPList(B);
[ 1 ]
[ a ]
[ a^2 ]
[ ab ]
```

To compute the image of right multiplication of the basis element `B[Length(B)]` of the module with the quotient algebra element corresponding to *ab* we use the function `MulQM` (3.9.4) with arguments `B[Length(B)]`, `GB2NP(a*b)`, and `GBR` We subsequently use `PrintNP` (3.2.1) to display the result as a 1-dimensional vector with an entry from *A*.

```
gap> v := MulQM(B[Length(B)],GP2NP(a*b),GBR);
[ [ [ -1 ] ], [ 1 ] ]
gap> PrintNP(v);
[ 1 ]
```

A.18 The dihedral group of order 8 on another module

In this example (Example 2 from Linton [Lin93]) the two-sided relations give the group algebra of the group with presentation $\langle a,b \mid a^4 = b^2 = (ab)^2 = 1\rangle$, the dihedral group of order 8. This module relation fixes the all-one vector of Example A.17: $1+a(1+a+b)$.

First load the package and set the standard infolevel `InfoGBNP` (4.2.1) to 0 and the time infolevel `InfoGBNPTime` (4.3.1) to 0 (for more information about the info level, see Chapter 4).

```
gap> LoadPackage("gbnp", false);
true
gap> SetInfoLevel(InfoGBNP,0);
gap> SetInfoLevel(InfoGBNPTime,0);
```

We will enter the relations as GAP polynomials. It is possible to enter these with and without a module. How to do this is shown in A.17. The relations here are entered without a module, since the module is only one-dimensional. It is possible to enter them using a free associative algebra with one over the field (the rational numbers) (see also **Reference: FreeAssociativeAlgebraWithOne for ring, rank (and name)**). For convenience we use the variables `a` and `b` for the generators of the algebra and `e` for the one of the algebra.

```
gap> A:=FreeAssociativeAlgebraWithOne(Rationals, "a", "b");
<algebra-with-one over Rationals, with 2 generators>
gap> g:=GeneratorsOfAlgebra(A);;
gap> a:=g[2];;b:=g[3];;e:=g[1];;
```

Now the relations are entered:

```
gap> twosidrels:=[a^4-e,b^2-e,(a*b)^2-e];;
gap> prefrels:=[ b-e, e + a * (e + a + b) ];;
```

First the relations are converted into NP format (see 2.1) after which the function `SGrobnerModule` (3.9.1) is called to calculate a Gröbner basis record.

```
gap> GBR:=SGrobnerModule(GP2NPList(prefrels),GP2NPList(twosidrels));;
```

The record GBR has two members: the two-sided relations `GBR.ts` and the prefix relations `GBR.p`. It is possible to print these using the function `PrintNPList` (3.2.3):

```
gap> PrintNPList(GBR.ts);
 b^2 - 1
 aba - b
 ba^2 - a^2b
 bab - a^3
 a^4 - 1
 a^3b - ba
gap> PrintNPList(GBR.p);
[ b - 1 ]
[ ab + a^2 + a + 1 ]
[ a^3 + a^2 + a + 1 ]
[ a^2b - a^2 ]
```

It is now possible to calculate the standard basis of the quotient module with the function BaseQM (3.9.2). This function has as arguments the Gröbner basis record GBR, the number of generators of the algebra (here it is 2), the number of generators of the mdoule (here it is 1), and a variable maxno for returning partial bases (0 means full basis).

```
gap> B:=BaseQM(GBR,2,1,0);;
gap> PrintNPList(B);
[ 1 ]
[ a ]
[ a^2 ]
```

A.19 The dihedral group on a non-cyclic module

In this example (Example 3 from Linton [Lin93]), the two-sided relations give the group algebra of the group with presentation $\langle a,b \mid a^4 = b^2 = (ab)^2 = 1\rangle$, the dihedral group of order 8. The module under construction is a non-cyclic module, obtained by taking two copies of the representation of Example A.17 and fusing their one-dimensional submodules.

Load the package and set the standard infolevel InfoGBNP (4.2.1) to 1 and the time infolevel InfoGBNPTime (4.3.1) to 1 (for more information about the info level, see Chapter 4).

```
gap> LoadPackage("gbnp", false);
true
gap> SetInfoLevel(InfoGBNP,1);
gap> SetInfoLevel(InfoGBNPTime,1);
```

Create the free associative algebra to enter the relations in:

```
gap> A:=FreeAssociativeAlgebraWithOne(Rationals, "a", "b");
<algebra-with-one over Rationals, with 2 generators>
gap> g:=GeneratorsOfAlgebra(A);;
gap> a:=g[2];;b:=g[3];;e:=g[1];;
```

Now the relations are entered:

```
gap> twosidrels:=[a^4-e,b^2-e,(a*b)^2-e];;
gap> D:=A^2;;
gap> y:=GeneratorsOfLeftModule(D);;
gap> modrels:=[y[1]*b-y[1], y[2]*b-y[2], y[1]+y[1]*a*(e+a+b) -y[2]-y[2]*a*(e+a+b)];;
```

First the relations are converted into NP format (see 2.1) with the function GP2NPList (3.1.2). They are printed in raw form and subsequently in a more legible format.

```
gap> modrelsNP:=GP2NPList(modrels);
[ [ [ [ -1, 2 ], [ -1 ] ], [ 1, -1 ] ], [ [ [ -2, 2 ], [ -2 ] ], [ 1, -1 ] ],
  [ [ [ -1, 1, 2 ], [ -1, 1, 1 ], [ -2, 1, 2 ], [ -2, 1, 1 ], [ -1, 1 ],
          [ -2, 1 ], [ -1 ], [ -2 ] ], [ 1, 1, -1, -1, 1, -1, 1, -1 ] ] ]
gap> PrintNPList(modrelsNP);
[ b - 1 , 0]
[ 0, b - 1 ]
[ ab + a^2 + a + 1 , - ab - a^2 - a - 1 ]
```

Next the function SGrobnerModule (3.9.1) is called to calculate a Gröbner basis record (see 2.8).

```
gap> GBR:=SGrobnerModule(modrelsNP,GP2NPList(twosidrels));;
#I  number of entered polynomials is 3
#I  number of polynomials after reduction is 3
#I  End of phase I
#I  End of phase II
#I  End of phase III
#I  End of phase IV
#I  The computation took 0 msecs.
#I  number of entered polynomials is 9
#I  number of polynomials after reduction is 9
#I  End of phase I
#I  End of phase II
#I  End of phase III
#I  End of phase IV
#I  The computation took 0 msecs.
```

The record GBR has two members: the two-sided relations GBR.ts and the prefix relations GBR.p. It is possible to print these using the function PrintNPList (3.2.3):

```
gap> PrintNPList(GBR.ts);
 b^2 - 1
 aba - b
 ba^2 - a^2b
 bab - a^3
 a^4 - 1
 a^3b - ba
gap> PrintNPList(GBR.p);
[ 0, b - 1 ]
[ b - 1 , 0]
[ ab + a^2 + a + 1 , - ab - a^2 - a - 1 ]
[ 0, a^3 - ab ]
[ 0, a^2b - a^2 ]
[ a^3 + a^2 + a + 1 , - ab - a^2 - a - 1 ]
[ a^2b - a^2 , 0]
```

It is now possible to calculate the standard basis of the quotient module with the function BaseQM (3.9.2). This function has as arguments the Gröbner basis record GBR, the number of generators of the algebra (in this case 2), the number of generators of the module (in this case 2), and a variable maxno for returning partial bases (0 means full basis).

```
gap> B:=BaseQM(GBR,2,2,0);;
gap> PrintNPList(B);
[ 0, 1 ]
[ 1 , 0]
[ 0, a ]
[ a , 0]
[ 0, a^2 ]
[ 0, ab ]
[ a^2 , 0]
```

It is also possible to convert each member of the list B of polynomials in NP form to GAP polynomials to do further calculations within the algebra or module. This can be done with the function NP2GPList (3.1.4).

```
gap> NP2GPList(B,D);
[ [ <zero> of ..., (1)*<identity ...> ], [ (1)*<identity ...>, <zero> of ... ]
    , [ <zero> of ..., (1)*a ], [ (1)*a, <zero> of ... ],
  [ <zero> of ..., (1)*a^2 ], [ <zero> of ..., (1)*a*b ],
  [ (1)*a^2, <zero> of ... ] ]
```

Individual GAP polynomials can be obtained from polynomials in NP form with the function NP2GP (3.1.3). This also holds for elements of the free module D in NP form.

```
gap> Display(NP2GP(B[Length(B)],D));
[ (1)*a^2, <zero> of ... ]
```

Next we write down the matrices for the right action of the generators on the module by means of MatrixQA (3.5.3).

```
gap> Display(MatrixQA(1,B,GBR));
[ [   0,   0,   1,   0,   0,   0,   0 ],
  [   0,   0,   0,   1,   0,   0,   0 ],
  [   0,   0,   0,   0,   1,   0,   0 ],
  [   0,   0,   0,   0,   0,   0,   1 ],
  [   0,   0,   0,   0,   0,   1,   0 ],
  [   1,   0,   0,   0,   0,   0,   0 ],
  [   1,  -1,   1,  -1,   1,   1,  -1 ] ]
gap> Display(MatrixQA(2,B,GBR));
[ [   1,   0,   0,   0,   0,   0,   0 ],
  [   0,   1,   0,   0,   0,   0,   0 ],
  [   0,   0,   0,   0,   0,   1,   0 ],
  [   1,  -1,   1,  -1,   1,   1,  -1 ],
  [   0,   0,   0,   0,   1,   0,   0 ],
  [   0,   0,   1,   0,   0,   0,   0 ],
  [   0,   0,   0,   0,   0,   0,   1 ] ]
```

In order to compute the image of the vector $2y[1]+3y[2]$ of the two standard generators of the module under the action of the element *aab*, we use StrongNormalFormNPM (3.9.5). Its first argument will be the vector and its second the Gröbner basis. The transformation GP2NP (3.1.1) to the NP format needs to be applied to the vector before it can be used as an argument.

```
gap> v:=StrongNormalFormNPM(GP2NP((y[1]*2+y[2]*3)*a*a*b), GBR);;
gap> PrintNP(v);
[ 2a^2 , 3a^2 ]
```

A.20 The icosahedral group

In this example the two-sided relations give the group algebra of the group with presentation $\langle a,b,c \mid a^2=b^2=c^2=(ab)^3=(bc)^5=(ac)^2=1\rangle$, the icosahedral group of order 120. This is the Coxeter group of type H_3. The module under construction is a 3-dimensional reflection representation,

First load the package and set the standard infolevel `InfoGBNP` (4.2.1) to 1 and the time infolevel `InfoGBNPTime` (4.3.1) to 1 (for more information about the info level, see Chapter 4).

```
gap> LoadPackage("gbnp", false);
true
gap> SetInfoLevel(InfoGBNP,1);
gap> SetInfoLevel(InfoGBNPTime,1);
```

Create the field containing the golden ratio `tau`.

```
gap> x := Indeterminate(Rationals,"x");
x
gap> p := x^2+ x-1;
x^2+x-1
gap> K := AlgebraicExtension(Rationals,p);
<algebraic extension over the Rationals of degree 2>
gap> tau:=RootOfDefiningPolynomial(K);
a
```

Create the free algebra with three generators over this field:

```
gap> A:=FreeAssociativeAlgebraWithOne(K, "a", "b", "c");
<algebra-with-one over <algebraic extension over the Rationals of degree
2>, with 3 generators>
gap> e:=One(A);; a:=A.a;; b:=A.b;; c:=A.c;;
```

The ideal for a quotient of the icosahedral group algebra over this field, in which `b*c` has a quadratic minimal polynomial involving `tau`:

```
gap> #(b*c)^2-tau*b*c+e
gap> Irels:=[a^2-e,b^2-e,c^2-e,a*b*a-b*a*b,((b*c)^2-tau*b*c+e)*(b*c-e),a*c-c*a];
[ (!-1)*<identity ...>+(!1)*a^2, (!-1)*<identity ...>+(!1)*b^2,
  (!-1)*<identity ...>+(!1)*c^2, (!1)*a*b*a+(!-1)*b*a*b,
  (!-1)*<identity ...>+(a+1)*b*c+(-a-1)*(b*c)^2+(!1)*(b*c)^3,
  (!1)*a*c+(!-1)*c*a ]
```

We now give module relations. The first two describe group elements of a vector stabilizer, the third forces the central element $(abc)^5$ to be nontrivial.

```
gap> Mrels:=[b*c-e,b-e,(a*b*c)^5+e];;
```

First the relations are converted into NP format (see 2.1) with the function `GP2NPList` (3.1.2). Next the function `SGrobnerModule` (3.9.1) is called to calculate a Gröbner basis record (see 2.8).

```
gap> GBR:=SGrobnerModule(GP2NPList(Mrels),GP2NPList(Irels));;
#I  number of entered polynomials is 6
#I  number of polynomials after reduction is 6
#I  End of phase I
#I  End of phase II
#I  End of phase III
#I  End of phase IV
```

```
#I  The computation took 16 msecs.
#I  number of entered polynomials is 12
#I  number of polynomials after reduction is 12
#I  End of phase I
#I  End of phase II
#I  End of phase III
#I  End of phase IV
#I  The computation took 36 msecs.
gap> PrintNPList(GBR.ts);;
 a^2 + !-1
 b^2 + !-1
 ca + !-1ac
 c^2 + !-1
 bab + !-1aba
 cbc + !-1bcb + -a-1c + a+1b
 bcba + !-1acba + !-1abcb + abac + cb + !-1bc + -a-2ba + a+2ab
 cbac + !-1acba + !-1abcb + abac + cb + !-1bc + !-1ba + -a-1ac + a+2ab
 bacba + abacb + !-1cba + !-1bcb + !-1abc + -a-2aba + c + a+2a
gap> PrintNPList(GBR.p);;
[ b + !-1 ]
[ c + !-1 ]
[ ac + !-1a ]
[ aba + !-1ab ]
[ abc + ab + -aa + -a ]
```

It is now possible to calculate the basis of the quotient algebra with the function `BaseQM` (3.9.2). This function has as arguments the Gröbner basis record `GBR`, the number of generators of the algebra (in this case 3), the number of generators of the free module in which the vectors are chosen (in this case 1), and a variable `maxno` for returning partial quotient algebras (0 means full basis).

```
gap> B:=BaseQM(GBR,3,1,0);;
gap> PrintNPList(B);
[ !1 ]
[ a ]
[ ab ]
```

Calculate the dimension of the quotient algebra with the function `DimQM` (3.9.3). This function has as arguments the Gröbner basis record `GBR`, the number of generators of the algebra (in this case 3) and the number of generators of the module (in this case 1).

```
gap> DimQM(GBR,3,1);
3
```

Next we write down the matrices for the right action of the generators on the module by means of `MatrixQA` (3.5.3).

```
gap> aa := MatrixQA(1,B,GBR);;
gap> Display(aa);
[ [  !0,  !1,  !0 ],
  [  !1,  !0,  !0 ],
  [  !0,  !0,  !1 ] ]
```

```
gap> bb := MatrixQA(2,B,GBR);;
gap> Display(bb);
[ [  !1,  !0,  !0 ],
  [  !0,  !0,  !1 ],
  [  !0,  !1,  !0 ] ]
gap> cc := MatrixQA(3,B,GBR);;
gap> Display(cc);
[ [   !1,   !0,   !0 ],
  [   !0,   !1,   !0 ],
  [    a,    a,  !-1 ] ]
```

Finally we check the defining relations for the icosahedral group on the three new matrix generators. This can be done by verifying if the result is equal to the identity matrix or with the function `IsOne` (**Reference: IsOne**).

```
gap> ee := IdentityMat(3,K);;
gap> Display(ee);
[ [  !1,  !0,  !0 ],
  [  !0,  !1,  !0 ],
  [  !0,  !0,  !1 ] ]
gap> aa^2 = ee;
true
gap> IsOne(aa^2);
true
gap> IsOne(bb^2);
true
gap> IsOne(cc^2);
true
gap> IsOne((aa*bb)^3);
true
gap> IsOne((aa*cc)^2);
true
gap> IsOne((bb*cc)^5);
true
```

A.21 The symmetric inverse monoid for a set of size four

The algebra we will consider is from Example 4 from Linton [Lin93]. Its monomials form the symmetric inverse monoid, that is, the monoid of all partial bijections on a given set, for a set of size four. The presentation is $\langle s_1, s_2, s_3, e \mid s_i^2 = (s_1s_2)^3 = (s_2s_3)^3 = (s_1s_3)^2 = 1, e^2 = e, s_1e = es_1, s_2e = es_2, es_3e = (es_3)^2 = (s_3e)^2\rangle$. The dimension of the monoid algebra is 209. The monoid has a natural representation of degree 4 by means of partial permutation matrices, which can be obtained by taking prefix relations $\{e, s_1 - 1, s_2 - 1, s_3e - s_3\}$.

First load the package and set the standard infolevel `InfoGBNP` (4.2.1) to 1 and the time infolevel `InfoGBNPTime` (4.3.1) to 1 (for more information about the info level, see Chapter 4).

```
gap> LoadPackage("gbnp", false);
true
gap> SetInfoLevel(InfoGBNP,1);
gap> SetInfoLevel(InfoGBNPTime,1);
```

Now enter the relations as GAP polynomials. The module is one dimensional so it is possible to enter it with and without a module. In Example 18 (A.17) both ways are shown. Here the relations will be entered without a module, with a free associative algebra with one over the field (the rational numbers) (see also **Reference: FreeAssociativeAlgebraWithOne for ring, rank (and name)**). For convenience we use the variables `s1`, `s2`, `s3`, and `e` for the generators of the algebra, and `o` for the identity element of the algebra.

```
gap> A:=FreeAssociativeAlgebraWithOne(Rationals, "s1", "s2", "s3", "e");
<algebra-with-one over Rationals, with 4 generators>
gap> g:=GeneratorsOfAlgebra(A);;
gap> s1:=g[2];;s2:=g[3];;s3:=g[4];;e:=g[5];;o:=g[1];;
```

It is possible to print symbols like they are printed in the algebra `A` with the function `GBNP.ConfigPrint` (3.2.2):

```
gap> GBNP.ConfigPrint(A);
```

Now the relations are entered:

```
gap> twosidrels:=[s1^2-o,s2^2-o,s3^2-o,(s1*s2)^3-o,(s2*s3)^3-o,(s1*s3)^2-o,
>   e^2-e,s1*e-e*s1,s2*e-e*s2,e*s3*e-(e*s3)^2,e*s3*e-(s3*e)^2];
[ (-1)*<identity ...>+(1)*s1^2, (-1)*<identity ...>+(1)*s2^2,
  (-1)*<identity ...>+(1)*s3^2, (-1)*<identity ...>+(1)*(s1*s2)^3,
  (-1)*<identity ...>+(1)*(s2*s3)^3, (-1)*<identity ...>+(1)*(s1*s3)^2,
  (-1)*e+(1)*e^2, (1)*s1*e+(-1)*e*s1, (1)*s2*e+(-1)*e*s2,
  (1)*e*s3*e+(-1)*(e*s3)^2, (1)*e*s3*e+(-1)*(s3*e)^2 ]
gap> prefixrels:=[e,s1-o,s2-o,s3*e-s3];
[ (1)*e, (-1)*<identity ...>+(1)*s1, (-1)*<identity ...>+(1)*s2,
  (-1)*s3+(1)*s3*e ]
```

First the relations are converted into NP format (see 2.1) and next the function `SGrobnerModule` (3.9.1) is called to calculate a Gröbner basis record.

```
gap> GBR:=SGrobnerModule(GP2NPList(prefixrels),GP2NPList(twosidrels));;
#I  number of entered polynomials is 11
#I  number of polynomials after reduction is 11
#I  End of phase I
#I  End of phase II
#I  End of phase III
#I  End of phase IV
#I  The computation took 24 msecs.
#I  number of entered polynomials is 42
#I  number of polynomials after reduction is 42
#I  End of phase I
#I  End of phase II
#I  End of phase III
#I  End of phase IV
#I  The computation took 20 msecs.
```

The record GBR has two members: the two-sided relations `GBR.ts` and the prefix relations `GBR.p`. We print these using the function `PrintNPList` (3.2.3):

```
gap> PrintNPList(GBR.ts);
 s1^2 - 1
 s2^2 - 1
 s3s1 - s1s3
 s3^2 - 1
 es1 - s1e
 es2 - s2e
 e^2 - e
 s2s1s2 - s1s2s1
 s3s2s3 - s2s3s2
 s3s2s1s3 - s2s3s2s1
 s3es3e - es3e
 es3es3 - es3e
 s3es3s2e - es3s2e
 s2s3s2es3e - s3s2es3e
 s3es3s2s1e - es3s2s1e
 es3s2es3s2 - es3s2es3
 s2s3s2s1es3e - s3s2s1es3e
 s2s3s2es3s2e - s3s2es3s2e
 s2es3s2es3e - es3s2es3e
 s1s2s1s3s2es3e - s2s1s3s2es3e
 s2s3s2s1es3s2e - s3s2s1es3s2e
 s2s3s2es3s2s1e - s3s2es3s2s1e
 s2es3s2s1es3e - es3s2s1es3e
 es3s2s1es3s2s1 - es3s2s1es3s2
 s1s2s1s3s2s1es3e - s2s1s3s2s1es3e
 s1s2s1s3s2es3s2e - s2s1s3s2es3s2e
 s1s2s1es3s2es3e - s2s1es3s2es3e
 s2s3s2s1es3s2s1e - s3s2s1es3s2s1e
 s2es3s2s1es3s2e - es3s2s1es3s2e
 s1s2s1s3s2s1es3s2e - s2s1s3s2s1es3s2e
 s1s2s1s3s2es3s2s1e - s2s1s3s2es3s2s1e
 s1s2s1es3s2s1es3e - s2s1es3s2s1es3e
 s1s3s2s1es3s2es3e - s3s2s1es3s2es3e
 s1s2s1s3s2s1es3s2s1e - s2s1s3s2s1es3s2s1e
 s1s2s1es3s2s1es3s2e - s2s1es3s2s1es3s2e
 s1s3s2s1es3s2s1es3e - s3s2s1es3s2s1es3e
 s1es3s2s1es3s2es3e - es3s2s1es3s2es3e
 s1s3s2s1es3s2s1es3s2e - s3s2s1es3s2s1es3s2e
gap> PrintNPList(GBR.p);
[ s1 - 1 ]
[ s2 - 1 ]
[ e ]
[ s3e - s3 ]
[ s3s2e - s3s2 ]
[ s3s2s1e - s3s2s1 ]
```

It is now possible to calculate the standard basis of the quotient module with the function `BaseQM` (3.9.2). This function has as arguments the Gröbner basis record `GBR`, the number of generators of the algebra (here this is 4), the number of generators of the module (here this is 1), and a variable `maxno` for returning partial bases (0 means full basis).

```
gap> B:=BaseQM(GBR,4,1,0);;
gap> PrintNPList(B);
[ 1 ]
[ s3 ]
[ s3s2 ]
[ s3s2s1 ]
```

Next we write down the matrices for the right action of the generators on the module. First by means of the list command `MatricesQA` (3.5.4), next one by one by means of `MatrixQA` (3.5.3) within a loop.

```
gap> MatricesQA(4,B,GBR);
[ [ [ 1, 0, 0, 0 ], [ 0, 1, 0, 0 ], [ 0, 0, 0, 1 ], [ 0, 0, 1, 0 ] ],
  [ [ 1, 0, 0, 0 ], [ 0, 0, 1, 0 ], [ 0, 1, 0, 0 ], [ 0, 0, 0, 1 ] ],
  [ [ 0, 1, 0, 0 ], [ 1, 0, 0, 0 ], [ 0, 0, 1, 0 ], [ 0, 0, 0, 1 ] ],
  [ [ 0, 0, 0, 0 ], [ 0, 1, 0, 0 ], [ 0, 0, 1, 0 ], [ 0, 0, 0, 1 ] ] ]
gap> for i in [1..4] do
>   Display(MatrixQA(i,B,GBR)); Print("\n");
> od;
[ [  1,  0,  0,  0 ],
  [  0,  1,  0,  0 ],
  [  0,  0,  0,  1 ],
  [  0,  0,  1,  0 ] ]

[ [  1,  0,  0,  0 ],
  [  0,  0,  1,  0 ],
  [  0,  1,  0,  0 ],
  [  0,  0,  0,  1 ] ]

[ [  0,  1,  0,  0 ],
  [  1,  0,  0,  0 ],
  [  0,  0,  1,  0 ],
  [  0,  0,  0,  1 ] ]

[ [  0,  0,  0,  0 ],
  [  0,  1,  0,  0 ],
  [  0,  0,  1,  0 ],
  [  0,  0,  0,  1 ] ]
```

A.22 A module of the Hecke algebra of type A_3 over GF(3)

This example is an extension of Example 5 from Linton, [Lin93]. It concerns the Hecke Algebra of type A_3. By reducing mod 3 but without evaluating at $q = 1$ it is possible to obtain the following representation of the Hecke algebra of type A_3 over GF(3): $\langle x,y,z \mid x^2+(1-q)*x-q, y^2+(1-q)*y-q, z^2+(1-q)*z-q, y*x*y-x*y*x, z*y*z-y*z*y, z*x-x*z\rangle$. It has a natural representation of the same dimension as the Lie algebra of type A_3, namely 4. This representation can be obtained with the module relations $\{x+1, y+1\}$.

First load the package and set the standard infolevel `InfoGBNP` (4.2.1) to 1 and the time infolevel `InfoGBNPTime` (4.3.1) to 1 (for more information about the info level, see Chapter 4).

```
gap> LoadPackage("gbnp", false);
true
gap> SetInfoLevel(InfoGBNP,1);
gap> SetInfoLevel(InfoGBNPTime,1);
```

Now enter the relations as GAP polynomials. The module is one dimensional so it is possible to enter it with and without a module. Both are used in Example A.17. Here the relations will be entered without using a module. First a free associative algebra with one is created over the field (GF(3)) (see also **Reference: FreeAssociativeAlgebraWithOne for ring, rank (and name)**). For convenience we use the variables `a` and `b` for the generators of the algebra and `e` for the one of the algebra.

```
gap> q:=Indeterminate(GF(3),"q");
q
gap> A:=FreeAssociativeAlgebraWithOne(Field(q), "x", "y", "z");;
gap> g:=GeneratorsOfAlgebra(A);;
gap> x:=g[2];;y:=g[3];;z:=g[4];;e:=g[1];;q:=q*e;;
```

In order to print the variables like they are printed in the algebra `A` with the function `GBNP.ConfigPrint` (3.2.2):

```
gap> GBNP.ConfigPrint(A);
```

Now the relations are entered:

```
gap> twosidrels:=[x^2+(e-q)*x-q,y^2+(e-q)*y-q,z^2+(e-q)*z-q,
>   y*x*y-x*y*x,z*y*z-y*z*y,z*x-x*z];
[ (-q)*<identity ...>+(-q+Z(3)^0)*x+(Z(3)^0)*x^2,
  (-q)*<identity ...>+(-q+Z(3)^0)*y+(Z(3)^0)*y^2,
  (-q)*<identity ...>+(-q+Z(3)^0)*z+(Z(3)^0)*z^2,
  (-Z(3)^0)*x*y*x+(Z(3)^0)*y*x*y, (-Z(3)^0)*y*z*y+(Z(3)^0)*z*y*z,
  (-Z(3)^0)*x*z+(Z(3)^0)*z*x ]
gap> prefixrels:=[x+e,y+e];
[ (Z(3)^0)*<identity ...>+(Z(3)^0)*x, (Z(3)^0)*<identity ...>+(Z(3)^0)*y ]
```

First the relations are converted into NP format (see 2.1) after which the function `SGrobnerModule` (3.9.1) is called to calculate a Gröbner basis record.

```
gap> GBR:=SGrobnerModule(GP2NPList(prefixrels),GP2NPList(twosidrels));;
#I  number of entered polynomials is 6
#I  number of polynomials after reduction is 6
#I  End of phase I
#I  End of phase II
#I  End of phase III
#I  End of phase IV
#I  The computation took 4 msecs.
#I  number of entered polynomials is 9
#I  number of polynomials after reduction is 9
#I  End of phase I
```

```
#I  End of phase II
#I  End of phase III
#I  End of phase IV
#I  The computation took 8 msecs.
```

The record GBR has three members: the two-sided relations `GBR.ts`, the prefix relations `GBR.p`, and the number `GBR.pg` of generators of the free module. It is possible to print the first two using the function `PrintNPList` (3.2.3):

```
gap> PrintNPList(GBR.ts);
 x^2 + -q+Z(3)^0x + -q
 y^2 + -q+Z(3)^0y + -q
 zx + -Z(3)^0xz
 z^2 + -q+Z(3)^0z + -q
 yxy + -Z(3)^0xyx
 zyz + -Z(3)^0yzy
 zyxz + -Z(3)^0yzyx
gap> PrintNPList(GBR.p);
[ x + Z(3)^0 ]
[ y + Z(3)^0 ]
```

It is now possible to calculate the standard basis of the quotient module with the function `BaseQM` (3.9.2). This function has as arguments the Gröbner basis record `GBR`, the number of generators of the algebra (here this is 3), the number of generators of the module (here this is 1), and a variable `maxno` for returning partial bases (0 means full basis).

```
gap> B:=BaseQM(GBR,3,1,0);;
gap> PrintNPList(B);
[ Z(3)^0 ]
[ z ]
[ zy ]
[ zyx ]
```

Next we write down the matrices for the right action of the generators on the module, by means of the command `MatricesQA` (3.5.4).

```
gap> MM := MatricesQA(3,B,GBR);;
gap> for i in [1..Length(MM)] do
> Display(MM[i]); Print("\n");
> od;
[ [   -Z(3)^0,    0*Z(3),    0*Z(3),    0*Z(3) ],
  [    0*Z(3),   -Z(3)^0,    0*Z(3),    0*Z(3) ],
  [    0*Z(3),    0*Z(3),    0*Z(3),    Z(3)^0 ],
  [    0*Z(3),    0*Z(3),         q,  q-Z(3)^0 ] ]

[ [   -Z(3)^0,    0*Z(3),    0*Z(3),    0*Z(3) ],
  [    0*Z(3),    0*Z(3),    Z(3)^0,    0*Z(3) ],
  [    0*Z(3),         q,  q-Z(3)^0,    0*Z(3) ],
  [    0*Z(3),    0*Z(3),    0*Z(3),   -Z(3)^0 ] ]

[ [    0*Z(3),    Z(3)^0,    0*Z(3),    0*Z(3) ],
```

```
[         q,  q-Z(3)^0,    0*Z(3),    0*Z(3) ],
[    0*Z(3),    0*Z(3),   -Z(3)^0,    0*Z(3) ],
[    0*Z(3),    0*Z(3),    0*Z(3),   -Z(3)^0 ] ]
```

A.23 Generalized Temperley-Lieb algebras

This example shows how the dimension of a Generalized Temperley-Lieb Algebra of type A, D, or E can be calculated. For A_{n-1} this is the usual Temperley-Lieb Algebra on n strands with dimension $\dim \mathrm{TL}(A_{n-1}) = \binom{2n}{n}/(n+1)$. For more information see [Gra95].

First load the package and set the standard infolevel `InfoGBNP` (4.2.1) to 0 and the time infolevel `InfoGBNPTime` (4.3.1) to 1 (for more information about timing; see Chapter 4).

```
gap> LoadPackage("gbnp", false);
true
gap> SetInfoLevel(InfoGBNP,0);
gap> SetInfoLevel(InfoGBNPTime,1);
```

The relations are generated automatically from the Coxeter diagram. This example can be easily adapted by specifying the number of points and the set of edges describing another Coxeter diagram. First enter the number of points, `numpoints`.

```
gap> numpoints:=8;
8
```

Now define some edges describing the diagrams of E_n, (these can be easily extended). In this example the dimension of the Generalized Temperley-Lieb algebra of type E_8 will be calculated. For $A_{1\ldots10}$ the command

`edges:=[[1,2],[2,3],[3,4],[4,5],[5,6],[6,7],[7,8],[8,9],[9,10]];`

can be used. For $D_{1\ldots10}$ the command

`edges:=[[1,3],[2,3],[3,4],[4,5],[5,6],[6,7],[7,8],[8,9],[9,10]];` can be used.

```
gap> edges:=[[1,3],[2,4],[3,4],[4,5],[5,6],[6,7],[7,8]]; # for E6..8
[ [ 1, 3 ], [ 2, 4 ], [ 3, 4 ], [ 4, 5 ], [ 5, 6 ], [ 6, 7 ], [ 7, 8 ] ]
```

Now enter the relations as GAP polynomials. First a free associative algebra with identity element is created over the Rationals (see also **Reference: FreeAssociativeAlgebraWithOne for ring, rank (and name)**). For convenience the generators are stored in `gens`.

```
gap> A:=FreeAssociativeAlgebraWithOne(Rationals,numpoints,"e");;
gap> e := GeneratorsOfAlgebraWithOne(A);
[ (1)*e.1, (1)*e.2, (1)*e.3, (1)*e.4, (1)*e.5, (1)*e.6, (1)*e.7, (1)*e.8 ]
```

It is possible to print symbols like they are printed in the algebra `A` with the function `GBNP.ConfigPrint` (3.2.2):

```
gap> GBNP.ConfigPrint(A);
```

Now the relations are generated automatically. For this we need to make sure the edges are sorted and converted to a set.

```
gap> edges:=Set(edges, x->SortedList(x));
[ [ 1, 3 ], [ 2, 4 ], [ 3, 4 ], [ 4, 5 ], [ 5, 6 ], [ 6, 7 ], [ 7, 8 ] ]
```

Now the relations can be generated. The relations are $e_i * e_i = e_i$, for all i and $e_i * e_j * e_i = e_i$ for all i,j that are connected in the Coxeter diagram and $e_i * e_j = e_j * e_i$ for all i, j that are not connected in the Coxeter diagram.

```
gap> rels:=[];;
gap> for i in [1..numpoints] do
>   for j in [1..numpoints] do
>     if (i=j) then
>       # if i=j then add e.i*e.i=e.i
>       Add(rels, e[i]*e[i]-e[i]);
>     elif ([i,j] in edges) or ([j,i] in edges) then
>       # if {i,j} is an edge then add e.i*e.j*e.i=e.i
>       Add(rels, e[i]*e[j]*e[i]- e[i]);
>     else
>       # if {i,j} is not an edge then add e.i*e.j=e.j*e.i
>       # (note: this causes double rules, but that's ok)
>       Add(rels, e[i]*e[j]- e[j]*e[i]);
>     fi;
>   od;
> od;
```

Then the relations are converted into NP format (see 2.1) after which the function `SGrobner` (3.4.2) is called to calculate a Gröbner basis.

```
gap> relsNP:=GP2NPList(rels);;
gap> GB:=SGrobner(relsNP);;
#I  The computation took 184 msecs.
```

It is now possible to calculate the dimension of the quotient algebra with the function `DimQA` (3.5.2). This function has as arguments the Gröbner basis `GB` and the number of generators of the algebra (here this is `numpoints`). To get the full basis the function `BaseQA` (3.5.1) can be used.

```
gap> DimQA(GB,numpoints);
10846
```

A.24 The universal enveloping algebra of a Lie algebra

Consider the Lie algebra with generators e, f and h, and relations $[e,f] = h$, $[e,h] = -2e$, $[f,h] = 2f$. This is the well-known Lie algebra of type A_1. We construct the corresponding universal enveloping algebra of this Lie algebra and show how one can prove that f^2 belongs to the ideal generated by e^2 in that associative algebra. The example is from Knopper's report [Kno04].

First load the package and set the standard infolevel `InfoGBNP` (4.2.1) to 0 and the time infolevel `InfoGBNPTime` (4.3.1) to 0 (for more information about the info level, see Chapter 4).

```
gap> LoadPackage("gbnp", false);
true
gap> SetInfoLevel(InfoGBNP,0);
gap> SetInfoLevel(InfoGBNPTime,0);
```

Then define the algebra and enter the relations as polynomials in GAP.

```
gap> A:=FreeAssociativeAlgebraWithOne(Rationals, "e", "f", "h");
<algebra-with-one over Rationals, with 3 generators>
gap> e:=A.e;; f:=A.f;; h:=A.h;; o:=One(A);;
gap> uerels:=[f*e-e*f+h,h*e-e*h-2*e,h*f-f*h+2*f];
[ (1)*h+(-1)*e*f+(1)*f*e, (-2)*e+(-1)*e*h+(1)*h*e, (2)*f+(-1)*f*h+(1)*h*f ]
```

The relations can be converted to NP format (see 2.1) with the function `GP2NPList` (3.1.2) and can be subsequently displayed with `PrintNPList` (3.2.3).

```
gap> uerelsNP:=GP2NPList(uerels);;
gap> PrintNPList(uerelsNP);
 ba - ab + c
 ca - ac - 2a
 cb - bc + 2b
```

Now configure printing in such a way that this algebra is used with the function `GBNP.ConfigPrint` (3.2.2).

```
gap> GBNP.ConfigPrint(A);
```

The set is actually a Gröbner basis, as can be verified by calculating the Gröbner basis with `SGrobner` (3.4.2).

```
gap> GB:=SGrobner(uerelsNP);;
gap> PrintNPList(GB);
 fe - ef + h
 he - eh - 2e
 hf - fh + 2f
```

Determine whether the quotient algebra is finite dimensional by means of `FinCheckQA` (3.6.2), with arguments the leading monomials of `GB` and 3, the number of variables involved. The leading monomials of `GB` are found by invoking `LMonsNP` (3.3.10).

```
gap> F:=LMonsNP(GB);
[ [ 2, 1 ], [ 3, 1 ], [ 3, 2 ] ]
gap> FinCheckQA(F,3);
false
```

Adding the relation $e^2 = 0$ results in a finite quotient algebra.

```
gap> extendedrels:=[f*e-e*f+h,h*e-e*h-2*e,h*f-f*h+2*f,e^2];
[ (1)*h+(-1)*e*f+(1)*f*e, (-2)*e+(-1)*e*h+(1)*h*e, (2)*f+(-1)*f*h+(1)*h*f,
  (1)*e^2 ]
gap> extendedrelsNP:=GP2NPList(extendedrels);;
```

With the function SGrobnerTrace (3.7.5) it is possible to calculate a Gröbner basis with trace information.

```
gap> GB:=SGrobnerTrace(extendedrelsNP);;
```

The Gröbner basis can now be displayed with PrintNPListTrace (3.7.4).

```
gap> PrintNPListTrace(GB);
 e^2
 eh + e
 fe - ef + h
 f^2
 fh - f
 he - e
 hf + f
 h^2 - 2ef + h
```

Note the fourth relation: $f^2 = 0$. To view a trace one can use the function PrintTracePol (3.7.3).

```
gap> PrintTracePol(GB[4]);
- 1/12G(1)f^2 + 1/12f^2G(1) + 1/12f^2G(1)h - 1/6fG(1)hf + 1/12G(1)hf^2 + 1/
24G(1)ef^3 + 1/24eG(1)f^3 - 1/8fG(1)ef^2 - 1/8feG(1)f^2 + 1/8f^2G(1)ef + 1/
8f^2eG(1)f - 1/24f^3G(1)e - 1/24f^3eG(1) - 1/24G(2)f^3 + 1/8fG(2)f^2 - 1/
8f^2G(2)f + 1/24f^3G(2) + 1/4G(3)f + 1/4fG(3) + 1/12fG(3)h + 1/12fhG(3) - 1/
12G(3)hf - 1/12hG(3)f - 1/12eG(3)f^2 + 1/6feG(3)f - 1/12f^2eG(3) + 1/24G(
4)f^4 - 1/6fG(4)f^3 + 1/4f^2G(4)f^2 - 1/6f^3G(4)f + 1/24f^4G(4)
```

This proves that $f^2 = 0$ is a consequence of $e^2 = 0$ in the universal enveloping algebra of the simple Lie algebra of type A_1.

The function StrongNormalFormTraceDiff (3.7.6) can be used to trace the difference between an element and its strong normal form in the terms of extendedrels. Apparently, in the first example the strong normal form of r is r - s.pol=0.

```
gap> r := [[[2,2,2,2,1,1,1,1]],[1]];;
gap> s := StrongNormalFormTraceDiff(r, GB);;

gap> PrintNP(s.pol);
 f^4e^4
gap> PrintTracePol(s);
 f^4G(4)e^2
gap> PrintNP(AddNP(r,s.pol,1,-1));
 0
```

One more example where the strong normal form is not zero.

```
gap> r := [[[3,3,3]],[1]];;
gap> s := StrongNormalFormTraceDiff(r, GB);;

gap> PrintNP(s.pol);
 h^3 - h
gap> PrintTracePol(s);
```

```
- G(1) - 1/2G(1)ef - 1/6eG(1)f + 1/3efG(1) + 1/2fG(1)e + 1/2feG(1) + G(
1)h^2 + 1/2G(1)efh + 1/2eG(1)fh + 1/3efG(1)h - 1/3eG(1)hf - 1/2fG(1)eh - 1/
2feG(1)h - 1/6eG(1)ef^2 - 1/6e^2G(1)f^2 + 1/3efG(1)ef + 1/3efeG(1)f - 1/
6ef^2G(1)e - 1/6ef^2eG(1) + 1/2G(2)f - 1/2fG(2) - 1/2G(2)fh + 1/2fG(2)h + 1/
6eG(2)f^2 - 1/3efG(2)f + 1/6ef^2G(2) - 2/3eG(3)h + 1/3ehG(3) + 1/3e^2G(3)f -
1/3efeG(3) - 1/2G(4)f^2 + fG(4)f - 1/2f^2G(4) + 1/2G(4)f^2h - fG(4)fh + 1/
2f^2G(4)h - 1/6eG(4)f^3 + 1/2efG(4)f^2 - 1/2ef^2G(4)f + 1/6ef^3G(4)
gap> PrintNP(AddNP(r,s.pol,1,-1));
 h
```

A.25 Serre's exercise

```
gap> LoadPackage("gbnp", false);
true
gap> SetInfoLevel(InfoGBNP,1);
gap> SetInfoLevel(InfoGBNPTime,1);
```

In Serre's book [Ser03] the following exercise can be found: Prove that the group $\langle \{a,b,c\} \mid \{bab^{-1}a^{-2}, cbc^{-1}b^{-2}, aca^{-1}c^{-2}\}\rangle$ is the trivial group. Here we solve the exercise by running the trace variant of the Gröbner basis function with input the set of equations $ba-a^2b, cb-b^2c, ac-c^2a$ together with relations forcing that a,b,c are invertible with inverse A,B,C.

```
gap> KI := [ [[2,1],[1,1,2]],[1,-1]],
>          [[3,2],[2,2,3]],[1,-1]],
>          [[1,3],[3,3,1]],[1,-1]],
>          [[1,4], []],[1,-1]],
>          [[4,1], []],[1,-1]],
>          [[2,5], []],[1,-1]],
>          [[5,2], []],[1,-1]],
>          [[3,6], []],[1,-1]],
>          [[6,3], []],[1,-1]],
>       ];;
gap> PrintNPList(KI);
 ba - a^2b
 cb - b^2c
 ac - c^2a
 ad - 1
 da - 1
 be - 1
 eb - 1
 cf - 1
 fc - 1
```

We use `SGrobnerTrace` (3.7.5), the trace variant of the Gröbner basis computation,

```
gap> GB := SGrobnerTrace(KI);;
#I  number of entered polynomials is 9
#I  number of polynomials after reduction is 9
#I  End of phase I
#I  End of phase II
```

```
#I  List of todo lengths is [ 9, 10, 11, 12, 14, 16, 18, 20, 21, 22, 24, 26, 28, 31, 34, 33, 35,
  43, 46, 49, 52, 56, 59, 62, 61, 60, 64, 64, 65, 65, 68, 71, 76, 76, 80, 88,
  93, 94, 99, 110, 115, 117, 131, 139, 146, 150, 158, 171, 186, 198, 206,
  220, 229, 246, 260, 263, 102, 40, 19, 9, 3, 0 ]
#I  End of phase III
#I  End of phase IV
#I  The computation took 19308 msecs.
```

The dimension of the quotient algebra is 1, showing that the group algebra is 1-dimensional. This implies that the group with the above presentation is trivial.

```
gap> GBpols := List([1..Length(GB)], x -> GB[x].pol);;
gap> PrintNPList(GBpols);
 a - 1
 b - 1
 c - 1
 d - 1
 e - 1
 f - 1
gap> DimQA(GBpols,6);
1
```

Since the output is large and might spoil the exercise, we confine ourselves to the printing first polynomial `GB[1]` and the length of its trace. As the trace polynomial expresses `GB[1]`, which is equal to $a-1$, as a combination of the binomials in `KI`, it gives a proof that a can be rewritten within the group to the trivial element. It is easy to derive from this that b and c are also trivial in the group. This justifies the restriction to `GB[1]`.

```
gap> PrintNP(GB[1].pol);
 a - 1
gap> Length(GB[1].trace);
1119
```

A.26 Baur and Draisma's transformations

```
gap> LoadPackage("gbnp", false);
true
gap> SetInfoLevel(InfoGBNP,0);
gap> SetInfoLevel(InfoGBNPTime,0);
```

The paper [BD04] by Baur and Draisma uses the computation of a quotient algebra of dimension 37, which we repeat here. The set of equations, after specialisation of the scalars to 1, is as follows.

```
gap> KI := [ [[[2,2]],[1]],
>          [[[1,1]],[1]],
>          [[[3,3]],[1]],
>          [[[1,2,1],[1]],[1,-1]],
>          [[[2,1,2],[2]],[1,-1]],
>          [[[3,2,3],[3]],[1,-1]],
```

```
>          [[[2,3,2],[2]],[1,-1]],
>          [[[1,3,1],[1]],[1,-1]],
>          [[[3,1,3],[3]],[1,-1]],
>          [[[1,2,3,1,2,3,1],[1,3,2,1,3,2,1],[1]],[1,1,-1]],
>          [[[3,1,2,3,1,2,3],[3,2,1,3,2,1,3],[3]],[1,1,-1]],
>          [[[2,3,1,2,3,1,2],[2,1,3,2,1,3,2],[2]],[1,1,-1]],
>        ];;
gap> PrintNPList(KI);
 b^2
 a^2
 c^2
 aba - a
 bab - b
 cbc - c
 bcb - b
 aca - a
 cac - c
 abcabca + acbacba - a
 cabcabc + cbacbac - c
 bcabcab + bacbacb - b
```

We carry out a traced Gröbner basis computation by use of `SGrobnerTrace` (3.7.5), and form the usual Gröbner basis by extracting the polynomials from the traced polynomials using the field indicator `.pol`.

```
gap> GBT := SGrobnerTrace(KI);;
gap> GB := List([1..Length(GBT)], i -> GBT[i].pol);;
```

The dimension of the quotient algebra is computable with `DimQA` (3.5.2).

```
gap> DimQA(GB,3);
37
```

In order to express the last GB element, viz.

```
gap> PrintNP(GB[Length(GB)]);
 cabcabca + cbacba - ca
```

as a combination of elements of `KI`, we use `PrintTracePol` (3.7.3):

```
gap> PrintTracePol(GBT[Length(GBT)]);
- G(9)bacba + cG(10)
```

We compute matrices for left multiplication by generators using `MatricesQA` (3.5.4) and determine the minimal polynomial of the sum of the three matrices.

```
gap> B := BaseQA(GB,3,0);;
gap> M := MatricesQA(3,B,GB);;
gap> f := MinimalPolynomial(Rationals,M[1]+M[2]+M[3]);
x_1^7-6*x_1^5+9*x_1^3-3*x_1
gap> Factors(f);
[ x_1, x_1^6-6*x_1^4+9*x_1^2-3 ]
```

It turns out that there are three non-zero numbers u, v, w such that the eigenvalues of the sum are $0, u, v, w, -u, -v, -w$. This is the information used in [BD04].

A.27 The cola gene puzzle

A prize question appearing in the January 2005 issue of the Dutch journal "Natuur en Techniek" asked for a DNA change of cows so that they could produce Cola instead of milk. A team of genetic manipulators has tools to perform the following five DNA string operations. (Here the strings before and after the equality sign can be interchanged at will.)

operation 1: TCAT = T;
operation 2: GAG = AG;
operation 3: CTC = TC;
operation 4: AGTA = A;
operation 5: TAT = CT.

The first question is to show how they can transform the milk gene TAGCTAGCTAGCT to the cola gene CTGACTGACT.

A second question is to show that mad cow disease related retro virus CTGCTACTGACT can be avoided at all times. Can this be guaranteed?

We answer these questions using the trace functions of the Gröbner basis package GBNP in Section 3.7.

First load the package and set the standard infolevel `InfoGBNP` (4.2.1) to 0 and the time infolevel `InfoGBNPTime` (4.3.1) to 0 to minimize the printing of data regarding the computation (for more information about the info level, see Chapter 4).

```
gap> LoadPackage("gbnp", false);
true
gap> SetInfoLevel(InfoGBNP,0);
gap> SetInfoLevel(InfoGBNPTime,0);
```

We introduce the free algebra `ALG` on the generators corresponding to the four letters in the DNA code and express the milk gene and cola gene as monomials in this algebra.

```
gap> ALG:=FreeAssociativeAlgebraWithOne(Rationals, "A", "C", "G", "T");;
gap> g:=GeneratorsOfAlgebra(ALG);;
gap> A:=g[2];;
gap> C:=g[3];;
gap> G:=g[4];;
gap> T:=g[5];;

gap> milk := T*A*G*C*T*A*G*C*T*A*G*C*T;;
gap> cola := C*T*G*A*C*T*G*A*C*T;;
```

We next enter the set K of binomials $x - y$ corresponding to the five operations $x = y$ listed above. We enter the binomials as members of `ALG` and let `KNP` be the corresponding set of NP polynomials.

```
gap> rule1 := T*C*A*T - T;;
gap> rule2 :=  G*A*G -  A*G;;
gap> rule3 :=  C*T*C - T*C;;
gap> rule4 := A*G*T*A - A;;
```

```
gap> rule5 := T*A*T -  C*T;;
gap> K := [rule1,rule2,rule3,rule4,rule5];;
gap> KNP := List(K,x-> GP2NP(x));;
```

We stipulate how the variables will be printed and print `KNP`. See `PrintNPList` (3.2.3).

```
gap> GBNP.ConfigPrint("A","C","G","T");
gap> PrintNPList(KNP);
 TCAT - T
 GAG - AG
 CTC - TC
 AGTA - A
 TAT - CT
```

Now calculate the usual Gröbner basis with `SGrobner` (3.4.2).

```
gap> GB := SGrobner(KNP);;
```

Compare milk and cola after taking their strong normal forms with respect to `GB` using `StrongNormalFormNP` (3.5.6). We observe that they have the same normal form and so there is a way to transform the milk gene into the cola gene.

```
gap> milkNP := GP2NP(milk);;
gap> colaNP := GP2NP(cola);;
gap> milkRed := NP2GP(StrongNormalFormNP(milkNP,GB),ALG);
(1)*T
gap> colaRed := NP2GP(StrongNormalFormNP(colaNP,GB),ALG);
(1)*T
```

But this information does not yet show us how to perform the transformation. To this end we calculate the Gröbner basis with trace information, using the function `SGrobnerTrace` (3.7.5).

```
gap> GTrace := SGrobnerTrace(KNP);;
```

The full trace can be printed with `PrintTraceList` (3.7.2), but we only print the relations (and no trace) by invoking `PrintNPListTrace` (3.7.4).

```
gap> PrintNPListTrace(GTrace);
 CT - T
 GA - A
 AGT - AT
 ATA - A
 TAT - T
 TCA - TA
```

In order to display a proof that $milk - cola$ belongs to the ideal we use `StrongNormalFormTraceDiff` (3.7.6), The result is a record, `p` say, containing `milk-cola` in the field `p.pol` (the normal form will be subtracted from the argument `milk-cola` to obtain `p.pol`, but the normal form is zero because the argument belongs to the ideal generated by `K`). The other field of the record `p` is `p.trace`, the traced polynomial, which is best displayed by use of `PrintTracePol` (3.7.3).

```
gap> p := StrongNormalFormTraceDiff(CleanNP(GP2NP(milk-cola)),GTrace);;
gap> NP2GP(p.pol,ALG);
(1)*(T*A*G*C)^3*T+(-1)*(C*T*G*A)^2*C*T
gap> PrintTracePol(p);
- TGATGAG(1) + TAGG(1)ATAT - TATATAGG(1) - TGAG(1)GACT + TGATGACG(1) - G(
1)GACTGACT - TAGCG(1)ATAT - TATAGG(1)AGT + TATATAGCG(1) + TGACG(1)GACT + CG(
1)GACTGACT - TAGG(1)AGTAGT + TAGTAGTAGG(1) + TATAGCG(1)AGT + TAGCG(
1)AGTAGT + TAGTAGG(1)AGCT - TAGTAGTAGCG(1) + TAGG(1)AGCTAGCT - TAGTAGCG(
1)AGCT - TAGCG(1)AGCTAGCT - TATG(2)TAT - TG(2)TATGAT - TGATGAG(3)AT + TAGG(
3)ATATAT - TATATAGG(3)AT - TGAG(3)ATGACT - G(3)ATGACTGACT - TATAGG(
3)ATAGT - TAGG(3)ATAGTAGT + TAGTAGTAGG(3)AT + TAGTAGG(3)ATAGCT + TAGG(
3)ATAGCTAGCT - TATG(4)T + TG(4)TAT + TATGG(4)T - TG(4)TGAT + TATATG(4)T + TGG(
4)TGAT - TG(4)TATAT + TATG(4)TAGT + TG(4)TAGTAGT + TAGG(5)ATAT - TATATAGG(
5) - TATAGG(5)AGT - TAGG(5)AGTAGT
```

In order to give a precise answer to the first question we need to work a little on `p.trace`. To do so, we introduce the following function, which creates the NP polynomial corresponding to the `i`-th term in the expression `p.trace` of `p.pol` as a linear combination of members of `KNP`. It is used to obtain the list `EvalList` of polynomials for all `i`.

```
gap> EvalTracePol := function(i,p,KNP)
>   local x,pi;
>   pi := p.trace[i];
>   x := BimulNP(pi[1],KNP[pi[2]],pi[3]);
>   return  [x[1],pi[4]*x[2]];
> end;;

gap> lev :=  Length(p.trace);;
gap> EvalList := List([1..lev], y -> CleanNP(EvalTracePol(y,p,KNP)));;
```

In order to find the rewrite from the milk gene to the cola gene as required for an answer to the first question, we match leading terms recursively.

```
gap> UnusedIndices := Set([1..lev]);;
gap> RunningTerm :=  milkNP[1][1];;
gap> stepno := 0;;
gap> NP2GP(milkNP,ALG);
(1)*(T*A*G*C)^3*T
gap> while Length(UnusedIndices) > 0 do
>   i := 0;
>   notfnd := true;
>   while i < lev and notfnd do
>     i := i+1;
>     if EvalList[i][1][1] = RunningTerm and i in UnusedIndices then
>        notfnd := false;
>        RemoveSet(UnusedIndices, i);
>        RunningTerm :=  EvalList[i][1][2];
>        stepno := stepno+1;
>     elif EvalList[i][1][2] = RunningTerm and i in UnusedIndices then
>        notfnd := false;
>        RemoveSet(UnusedIndices, i);
```

```
>        RunningTerm :=  EvalList[i][1][1];
>        stepno := stepno+1;
>     fi;
>   od;
>   if i = lev and notfnd = true then Print("error not fnd in"); fi;
>   Print(" -(",stepno,")- ");
>   PrintNP([[p.trace[i][1]],[1]]);
>   Print("         K[",p.trace[i][2],"]\n        ");
>   PrintNP([[p.trace[i][3]],[1]]);
>   Print(" --> ");
>   PrintNP([[EvalList[i][1][2]],[1]]);
> od;;
 -(1)-  TAGC
         K[1]
         AGCTAGCT
 -->  TAGCTAGCTAGCT
 -(2)-  TAG
         K[3]
         ATAGCTAGCT
 -->  TAGTCATAGCTAGCT
 -(3)-  TAG
         K[1]
         AGCTAGCT
 -->  TAGTAGCTAGCT
 -(4)-  TAGTAGC
         K[1]
         AGCT
 -->  TAGTAGCTAGCT
 -(5)-  TAGTAG
         K[3]
         ATAGCT
 -->  TAGTAGTCATAGCT
 -(6)-  TAGTAG
         K[1]
         AGCT
 -->  TAGTAGTAGCT
 -(7)-  TAGTAGTAGC
         K[1]
         1
 -->  TAGTAGTAGCT
 -(8)-  TAGTAGTAG
         K[3]
         AT
 -->  TAGTAGTAGTCAT
 -(9)-  TAGTAGTAG
         K[1]
         1
 -->  TAGTAGTAGT
 -(10)-  TAG
         K[1]
         AGTAGT
 -->  TAGTAGTAGT
```

```
-(11)-  TAG
        K[3]
        ATAGTAGT
-->  TAGTCATAGTAGT
-(12)-  TAGC
        K[1]
        AGTAGT
-->  TAGCTAGTAGT
-(13)-  TAG
        K[5]
        AGTAGT
-->  TAGCTAGTAGT
-(14)-  T
        K[4]
        TAGTAGT
-->  TATAGTAGT
-(15)-  TATAG
        K[1]
        AGT
-->  TATAGTAGT
-(16)-  TATAG
        K[3]
        ATAGT
-->  TATAGTCATAGT
-(17)-  TATAGC
        K[1]
        AGT
-->  TATAGCTAGT
-(18)-  TATAG
        K[5]
        AGT
-->  TATAGCTAGT
-(19)-  TAT
        K[4]
        TAGT
-->  TATATAGT
-(20)-  TATATAG
        K[1]
        1
-->  TATATAGT
-(21)-  TATATAG
        K[3]
        AT
-->  TATATAGTCAT
-(22)-  TATATAGC
        K[1]
        1
-->  TATATAGCT
-(23)-  TATATAG
        K[5]
        1
-->  TATATAGCT
```

```
-(24)-  TATAT
        K[4]
        T
-->  TATATAT
-(25)-  T
        K[4]
        TATAT
-->  TATATAT
-(26)-  TAG
        K[5]
        ATAT
-->  TAGCTATAT
-(27)-  TAGC
        K[1]
        ATAT
-->  TAGCTATAT
-(28)-  TAG
        K[3]
        ATATAT
-->  TAGTCATATAT
-(29)-  TAG
        K[1]
        ATAT
-->  TAGTATAT
-(30)-  T
        K[4]
        TAT
-->  TATAT
-(31)-  TAT
        K[4]
        T
-->  TATAT
-(32)-  TAT
        K[2]
        TAT
-->  TATAGTAT
-(33)-  TATG
        K[4]
        T
-->  TATGAT
-(34)-  T
        K[4]
        TGAT
-->  TATGAT
-(35)-  T
        K[2]
        TATGAT
-->  TAGTATGAT
-(36)-  TG
        K[4]
        TGAT
-->  TGATGAT
```

```
  -(37)-  TGATGA
          K[1]
          1
  -->  TGATGAT
  -(38)-  TGATGA
          K[3]
          AT
  -->  TGATGATCAT
  -(39)-  TGATGAC
          K[1]
          1
  -->  TGATGACT
  -(40)-  TGA
          K[1]
          GACT
  -->  TGATGACT
  -(41)-  TGA
          K[3]
          ATGACT
  -->  TGATCATGACT
  -(42)-  TGAC
          K[1]
          GACT
  -->  TGACTGACT
  -(43)-  1
          K[1]
          GACTGACT
  -->  TGACTGACT
  -(44)-  1
          K[3]
          ATGACTGACT
  -->  TCATGACTGACT
  -(45)-  C
          K[1]
          GACTGACT
  -->  CTGACTGACT
gap> NP2GP(colaNP,ALG);
(1)*(C*T*G*A)^2*C*T
```

And now the second question regarding the retro virus.

```
gap> retro := C*T*G*C*T*A*C*T*G*A*C*T;;
```

We compute the Strong Normal Form `StrongNormalFormNP` (3.5.6) of `retro` with respect to `GB`. As it is `TGT`, distinct to `T`, the strong normal form of milk, there is no transformation from milk to retro.

```
gap> NP2GP(StrongNormalFormNP(CleanNP(GP2NP(retro)),GB), ALG);
(1)*T*G*T
```

Of course, here too we can verify the reduction, by computing `StrongNormalFormTraceDiff` (3.7.6) with input the NP polynomial corresponding to `retro` and with respect to `K`; it is called `retroTrace`.

The symbol G in expression like G(2) are not to be confused with the single symbols G representing the DNA element.

```
gap> retroTrace := StrongNormalFormTraceDiff(CleanNP(GP2NP(retro)),GTrace);;
gap> PrintTracePol(retroTrace);
 TGG(1) - TGC^2G(1) - TGTAG(1) + TGTACG(1) + TGTAGG(1)AT + TGTATGAG(
1) + TGTAG(1)GACT - TGTAGCG(1)AT - TGTATGACG(1) + TGG(1)ACTGACT - TGTACG(
1)GACT + G(1)GCTACTGACT - TGCG(1)ACTGACT - CG(1)GCTACTGACT + TGTATG(
2)TAT + TGG(3)AT + TGCG(3)AT - TGTAG(3)AT + TGTAGG(3)ATAT + TGTATGAG(
3)AT + TGTAG(3)ATGACT + TGG(3)ATACTGACT + G(3)ATGCTACTGACT + TGTG(
4)T + TGTATG(4)T - TGTG(4)TAT - TGTATGG(4)T + TGCG(5) + TGG(5)AT - TGTAG(
5) + TGTAGG(5)AT
```

References

[BD04] Karin Baur and Jan Draisma. Higher secant varieties of the minimal adjoint orbit. *J. Algebra*, 280(2):743–761, 2004. 120, 122

[CCS99] Arjeh M. Cohen, Hans Cuypers, and Hans Sterk. *Some Tapas of Computer Algebra*, volume 4 of *Algorithms and Computation in Mathematics*. Springer-Verlag, Heidelberg, 1999. 56, 78, 96

[CGW05] Arjeh M. Cohen, Dié A. H. Gijsbers, and David B. Wales. BMW algebras of simply laced type. *J. Algebra*, 286(1):107–153, 2005. 80

[CLO97] David Cox, John Little, and Donal O'Shea. *Ideals, varieties, and algorithms*. Undergraduate Texts in Mathematics. Springer-Verlag, New York, second edition, 1997. An introduction to computational algebraic geometry and commutative algebra. 55

[Coh07] Arjeh M. Cohen. Non-commutative polynomial computations. Report, 2007. Eindhoven. 6, 10, 25, 26, 27, 29

[Con10] Randall Cone. NMODoc - documentation on the nmo package, version 1.0. Report (the package is provided with GBNP from Version 1.0 on), Jan 2010. 2, 8

[Gra95] John J. Graham. *Modular representations of Hecke algebras and related algebras.* PhD thesis, The University of Sydney, Sep 1995. 115

[Gre99] Edward L. Green. Noncommutative Gröbner bases and projective resolutions. In *Computational Methods for Representations of Groups and Algebras*, pages 29–60. Birkhäuser, 1999. (Essen 1997). 2, 51, 53

[Kno04] Jan Willem Knopper. GBNP and vector enumeration. Report, 2004. Eindhoven. 6, 116

[Kro03] Chris Krook. Dimensionality of quotient algebras. Report, 2003. Eindhoven. 2, 6, 36, 93, 94

[Lin93] Steve A. Linton. On vector enumeration. *Linear Algebra Appl.*, 192:235–248, 1993. Computational linear algebra in algebraic and related problems (Essen, 1992). 100, 103, 104, 109, 112

[LN06] Frank Lübeck and Max Neunhöffer. GAPDoc - a GAP package, version 0.99999, Jan 2006. 2

[Mor94] Teo Mora. An introduction to commutative and noncommutative Gröbner bases. *Theoretical Computer Science*, 134(1):131–173, Nov 1994. 2, 51, 88, 90

[NR02] Gonzalo Navarro and Mathieu Raffinot. *Flexible Pattern Matching in Strings*. Cambridge University Press, 2002. 63

[Ser03] Jean-Pierre Serre. *Trees*. Springer Monographs in Mathematics. Springer-Verlag, Berlin, 2003. Translated from the French original by John Stillwell, Corrected 2nd printing of the 1980 English translation. 119

[Ufn89] V. A. Ufnarovskiĭ. On the use of graphs for calculating the basis, growth and Hilbert series of associative algebras. *Mat. Sb.*, 180(11):1548–1560, 1584, 1989. 2

Index