

TECHNISCHE UNIVERSITEIT EINDHOVEN

Department of Mathematics and Computing Science

Internship Report

GBNP and Vector Enumeration

by
J.W. Knopper

Supervisor: prof.dr. Arjeh M. Cohen
Supervisor: dr. Steve A. Linton
Eindhoven, October 2004

Contents

1	Introduction	7
2	Gröbner bases	9
2.1	Introduction	9
2.2	Definitions	9
2.3	Outline of the algorithm	10
2.3.1	Start of the algorithm	11
2.3.2	Algorithm step	11
2.3.3	End of the algorithm	11
2.3.4	About termination	11
2.4	Modules	11
2.4.1	Introduction	11
2.4.2	Ordering	12
2.4.3	Using the original algorithm for modules	12
3	Vector enumerator	15
3.1	Introduction	15
3.2	Definitions	15
3.3	Table	15
3.4	Adding rows to the table	15
3.5	Applying the relators	16
3.6	Pressing coincidences	16
3.7	Starting and finishing conditions	16
3.8	Comparison	16
4	Making GBNP a GAP-package	17
4.1	Introduction	17
4.2	Documentation	17
4.2.1	GAPDoc	17
4.2.2	GAPDoc:Include	18
4.2.3	GAPDoc:Functions	18
4.2.4	GAPDoc:Examples	19
4.2.5	GAPDoc:References	20
4.2.6	Producing the GAPDoc documentation	21
4.2.7	README	21
4.2.8	PackageInfo.g	21
4.3	Splitting declaration and implementation	22
4.4	Function names	22
4.4.1	GBNP-record	22
4.4.2	Global Variables	23
4.5	Better interworking	23
4.6	Verbose Functions	23

4.7	Tests	23
4.8	Printing symbols	23
5	Some improvements in the implementation	25
5.1	Introduction	25
5.2	Occur	25
5.3	RightOccur	25
5.4	RightOccurInLst, RightOccur tree-structure	26
5.4.1	Recursive definition	26
5.4.2	An example	27
5.4.3	Creating a tree	27
5.4.4	LeftOccur-trees	27
5.5	Other uses for OccurTrees	28
5.5.1	Introduction	28
5.5.2	Using trees in OccurInLst	28
5.5.3	Obstruction searching	28
5.5.4	Removing elements and sorting arrays	29
5.5.5	Extra arrays tree2arr and arr2tree	29
5.6	Other improvements	29
5.6.1	Boolean lists	29
5.6.2	Using a merge sort when adding	30
5.7	Further suggestions	30
5.7.1	Data structures	30
5.7.2	PositionSublist	30
5.7.3	Other places to speed up	30
5.7.4	Compiling	31
5.7.5	Partial prefix basis	31
5.7.6	Combining variants	31
5.7.7	Optimisations where needed	31
6	Application: Lie algebras	33
6.1	Introduction	33
6.2	Introduction to Lie algebras	33
6.3	Construction	34
6.3.1	Introduction	34
6.3.2	Construction	34
6.3.3	Representations where only some root elements are quadratic	35
6.3.4	GF(2)/GF(3)	35
6.3.5	Expected results	35
6.3.6	Lie algebras in other algebras	36
6.4	Results	36
6.4.1	GF(2)	36
6.4.2	GF(3)	37
6.5	Meataxe	37
A	GAP-specific issues	39
A.1	Introduction	39
A.2	Profiling	39
A.2.1	Introduction	39
A.2.2	GAP profiling functions	39
A.2.3	Example	39
A.2.4	Profiling record functions	40
A.3	Methods and objects	40
A.4	Demo Mode	41

A.5	Assertions	41
A.6	Compiling	41
A.7	Functions and different argument lengths	41
A.8	SetHelpViewer	41
A.9	Set additional GAP home directories	42

Chapter 1

Introduction

In Eindhoven the computer algebra package GBNP (Gröbner basis for non-commutative polynomials) has been developed by Dié A. H. Gijssbers and Arjeh M. Cohen. This computer algebra package makes it possible, given an algebra by means of its generators and relations, to find a matrix representation of the same algebra (if finite dimensional) (see also <http://www.win.tue.nl/~amc/pub/grobner/doc.html>). Information about Gröbner bases can be found in Chapter 2.

During an internship at the University of St. Andrews, a part of my 5 year study program of Technical Mathematics at the Eindhoven University of Technology, I worked on and with this package. This paper is the result of that internship. My guide in Eindhoven was Arjeh M. Cohen, and my guide in St. Andrews was Steve A. Linton, who is the maintainer of GAP, which stands for Groups, Algorithms and Programming, and has written some articles about vector enumeration. Vector enumeration will be explained briefly in Chapter 3.

I have developed the package further, extending the package to allow use of arbitrary fields and modules. More details about modules can be found in Section 2.4. Furthermore it is now possible to make the package available as a part (share package) of GAP. More about this can be found in Chapter 4.

Some significant speed improvements in the implementation have been obtained by the use of a particular data structure, see Section 5.4. A simpler version of this data structure has already been used by Chris Krook [5], but it was not clear if and how this data structure could be used to make further improvements. Also some other improvements were made. These improvements are described in Chapter 5.

Chapter 2

Gröbner bases

2.1 Introduction

This chapter contains a brief introduction to non-commutative Gröbner bases. It follows the introduction from the proof of the correctness of the algorithm used by GBNP, written by the authors of GBNP, Arjeh M. Cohen and Dié A.H. Gijssbers[1]. This differs only slightly from the notation used by Teo Mora[8].

First we discuss some definitions. In Section 2.2 we discuss two-sided ideals as used in GBNP. In Section 2.4 we discuss some study about how to combine the right ideals with the two-sided ideals and study the module of a quotient algebra.

2.2 Definitions

Let k be a field and T be a free monoid on n generators x_1, \dots, x_n . An element of the monoid ring $k\langle T \rangle$ is called a non-commutative polynomial or sometimes polynomial.

Gröbner bases are about unique reduction: the order of the reduction does not matter. To be able to speak about a reduction, we first have to define an ordering:

Definition 2.1. *An ordering $<$ on T is called a reduction ordering if for all $t_1, t_2, l, r \in T$, with $t_1 < t_2$ we have $1 \leq lt_1r < lt_2r$.*

The reduction ordering which is used in the package GBNP is “total degree first, then lexicographic” (Mora calls this the *deglex* ordering).

It is now possible to write a polynomial in an ordered way.

Definition 2.2. *Given an ordering on T it is possible to write each polynomial $f \in k\langle T \rangle$ in a unique way as a linear combination of monomials t_i :*

$$f = \sum_{i=1}^s c_i t_i$$

with $c_i \in k \setminus \{0\}$ and $t_i \in T$, such that $t_1 > \dots > t_s$. This decomposition is called the ordered form of f . The polynomial f is called monic if $c_1 = 1$. The largest monomial with nonzero coefficient of f , t_1 , is denoted as $L(f)$ and called the leading term.

Given a list of polynomials known to be zero, a polynomial can be reduced to a smaller polynomial modulo the polynomials in the list. With a reduction ordering this process ends in a finite number of steps. The resulting polynomial is not unique in general. The order of the reduction might influence the result. For a Gröbner basis the polynomial that cannot be reduced further is always the same. In that case the reduction is unique:

Definition 2.3. Let $G \subset k\langle T \rangle$, and denote by I the ideal generated by G . A normal form of $f \in k\langle T \rangle$ with respect to G is an element $h \in k\langle T \rangle$ such that $f - h \in I$ and either $h = 0$ or $L(g) \nmid L(h)$ for all $g \in G$.

Moreover, G is a Gröbner basis of I if G is a basis of I and if 0 is a normal form with respect to G of each element of I .

Let G be a basis for an ideal I . If G is not a Gröbner basis then some words in the ideal I do not have normal form 0 . They can be found by generating new relations of the relations in G .

Definition 2.4. Let $G = (g_i)_{1 \leq i \leq n}$ be a list of monic polynomials. An obstruction of G is a six-tuple $(l, i, r; \lambda, j, \rho)$ with $i, j \in \{1, \dots, n\}$ and $l, \lambda, r, \rho \in T$ such that $L(g_i) \leq L(g_j)$ and $lL(g_i)r = \lambda L(g_j)\rho$. Given an obstruction we define the corresponding S-polynomial as

$$s(l, i, r; \lambda, j, \rho) = lg_i r - \lambda g_j \rho$$

If a relation does not add anything new, then it does not need to be added to G . Such is the case with weak obstructions:

Definition 2.5. Given a list $G = (g_i)_{1 \leq i \leq k}$ of monic polynomials a polynomial f is called weak with respect to G if there are $c_h \in k$ and $l_h, r_h \in T$ such that $l_h L(g_h) r_h \leq L(f)$ and

$$f = \sum_h c_h l_h g_h r_h.$$

An obstruction $(l, i, r; \lambda, j, \rho)$ of G is called weak if its S-polynomial $s(l, i, r; \lambda, j, \rho)$ is weak with respect to G .

It is possible to decrease the number of useful obstructions even further. The obstructions can also be used to reduce each other. The reader is referred to [1] for details. A set of polynomials that is large enough to contain the same information as the total set of instructions is called a basic set.

Lemma 2.1 (Lemma 2.2 from [1]). There is a finite basic set H of S-polynomials of G . Moreover, H can be chosen so that every member of H is an S-polynomial of G with overlap and with at least one of the two parameters $\{l, \lambda\}$ and one of $\{r, \rho\}$ equal to 1.

It is now possible to distinguish between three kinds of obstruction:

Definition 2.6. Let $s = (l, i, r; \lambda, j, \rho)$ be an obstruction of the list G of monic polynomials in $k\langle T \rangle$.

- If $l = 1$ then s is called a right obstruction ($s = (1, i, 1; \lambda, j, \rho)$ or $s = (1, i, r; \lambda, j, 1)$).
- If $r = 1$ and $l \neq 1$ then s is called a left obstruction ($s = (l, i, 1; 1, j, \rho)$).
- The remaining obstructions with $\lambda = \rho = 1$ are called central obstructions ($s = (l, i, r; 1, j, 1)$).

2.3 Outline of the algorithm

The algorithm is only described briefly here and can be found in detail in [1].

Let I be a two-sided ideal of $k\langle T \rangle$ and let G, D be finite subsets of $k\langle T \rangle$. Suppose G is a basis for I and D is a basic set for G .

The two-sided algorithm basically moves elements from D to G until D is empty. G is then a Gröbner basis.

2.3.1 Start of the algorithm

To start a computation initialize as follows:

- Let G be list of polynomials, after some “cleaning operations” to fulfill an invariant, which is not described here.
- Let D be the normal forms of the left, right and central obstructions of G (this is a basic set).
- Some cleaning is also needed to fulfill an additional requirement.

2.3.2 Algorithm step

1. One polynomial is moved from D to G .
2. The left, right and central obstructions of that polynomial are calculated, reduced and added to D so that D stays basic for the new G .
3. Check if the polynomials in G can be reduced and if they can, reduce them and add new S-polynomials to D .
4. Reduce all polynomials in D .

2.3.3 End of the algorithm

The algorithm ends if no more obstructions need to be added:

Theorem 2.1 (Theorem 3.1 from [1]). *If D is the empty set, then G is a Gröbner basis for I .*

2.3.4 About termination

About the original Buchberger algorithm the following is known:

Lemma 2.2 (adapted from [3], Proposition 2.8). *If the (two-sided) ideal of leading terms has a finite number of monomial generators, then the algorithm terminates in a finite number of steps and yields a finite Gröbner basis.*

This means that if a finite Gröbner basis exists, its leading terms will generate the two-sided ideal of leading terms and therefore the algorithm will end. The original algorithm also adds a lot of weak obstructions.

Note that not all ideals have a finite number of monomial generators because of the word problem. Furthermore it may depend on the ordering whether or not a finite Gröbner basis exists.

2.4 Modules

2.4.1 Introduction

The vector enumerator gives a representation of a right-module. In this section it will be described how similar functionality has been added to GBNP.

In this section all modules are right modules. Let $G \subset k\langle T \rangle$ be a basis for an ideal $I = \langle AGA \rangle$ in a free noncommutative algebra A . Let P be A/I . Let A^s be a vector space of dimension s over A . Let $W \subseteq A^s$ be a set of relations. It is sometimes possible to compute $(A/I)^s / \overline{W}$, where \overline{W} is the image of W in $(A/I)^s$.

2.4.2 Ordering

There are several possibilities for a reduction ordering for modules. In this paragraph the discussion will be mainly about how the ordering used in the original algorithm (first degree, then lexicographic) can be extended to modules.

Except for degree and lexicographical order there is now another way of ordering monomials: by means of the generator of the module. It seems logical to use the reverse order for this (so that for e_1, e_2 standard generators of the module A^2 and m a monomial in the algebra, $(m, 0) = e_1 * m > e_2 * m = (0, m)$).

It is possible to use the order of the generators of the module, as the dominant part of the ordering, after the degree, or as the least dominant ordering:

- module generators, degree, lexicographic. This ordering does not always result in the shortest words, but it is a reduction ordering, because it is possible to assign weights to the module generators for each finitely generated ideal. The advantage of this ordering is that it is easy to split polynomials up into groups with the same module generator.
- degree, module generators, lexicographic. Monomials of the same length are first sorted by module generator and then lexicographically.
- degree, lexicographic, module generators. This ordering might be the most natural, but it is a bit harder to implement. It might be desired to change the ordering functions so that this can be done in the future.

The ordering from the original package can be extended to the second ordering with no changes in the implementation. It is possible to use the last ordering but this would be less efficient in the current representation.

2.4.3 Using the original algorithm for modules

We will now show that it is possible to use the original Gröbner basis algorithm after a transformation.

Let k be a field and A be the algebra $k\langle T \rangle$. Let $M = \{m_1, \dots, m_s\}$ be a set of generators of a free A -module. Write $F = k\langle T \cup M \rangle$ and $G = k\langle T \cup M \cup \{e\} \rangle$. Note the free A -module generated by M is a submodule of the A -module on F . This may be viewed as a left A -module.

Let $R \subseteq A \subseteq F$ be a set of relations. Let $W \subseteq A^s \subseteq F$ be a set of A -module relations. Let $W_e = \{x \cdot e - x \mid x \in T \cup M \cup \{e\}\} \cup \{e \cdot t - t \mid t \in T\}$. Let $W_M = \{x \cdot m_i \mid x \in T \cup M \cup \{e\}, m_i \in M\}$. Note that W_e contains the relation $e^2 - e$, so e is an *idempotent* and that $W_e, W_M \subset F$.

Theorem 2.2. $F/\langle FW_M F \rangle \cong A \oplus m_1 A \oplus \dots \oplus m_s A$ as free A -algebras of rank $s + 1$.

Proof. Let x be a monomial in G . It is possible that $x = 1$, in which case $x \in A$. Suppose $x \neq 1$.

A monomial $x \in G$ can fall into three categories, when it comes to containing m_i .

- x does not contain any $m \in M$. Then $x \in A$.
- x contains any $m \in M$, which is not at the left of x . In this case it is possible to reduce x to zero with the rules in W_M .
- x contains one $m \in M$, and does contain that m at the left. Then $x \in m_1 A \oplus \dots \oplus m_s A$ and more specifically $x \in mA$.

Multiplication of two monomials a and b is as follows:

- if $a, b \in A$ then $a \cdot b \in A$.
- if $a \in A, a \neq 1$ and $b \in m_1 A \oplus \dots \oplus m_s A$ then $a \cdot b = 0$ (after reduction with W_M).
- if $a = 1 \in A$ and $b \in m_1 A \oplus \dots \oplus m_s A$ then $a \cdot b = b$.

- if $a \in m_1A \oplus \cdots \oplus m_sA$ and $b \in A$ then $a \cdot b \in m_1A \oplus \cdots \oplus m_sA$.
- if $a, b \in m_1A \oplus \cdots \oplus m_sA$ then $a \cdot b = 0$ (after reduction with W_M).

□

Because the special case $a = 1 \in A$ is not desired we add an idempotent e to F , forming G .

Theorem 2.3. $G/\langle GW_eG, GW_MG \rangle \cong (A + ek) \oplus m_1A \oplus \cdots \oplus m_sA$ as A -algebras where $A + ek$ is the A -module spanned by a free submodule A and an additional generator e satisfying $e \cdot t = t$, for $t \in T$.

Proof. Let x be a monomial in G . It is possible that $x = 1$ or $x = e$, in which in both cases $x \in A + ek$. Suppose $x \neq 1, e$. If x contains e then it is possible to reduce it with the rules in W_e . So we may suppose that x does not contain e .

A monomial $x \in G$ can fall into three categories, when it comes to containing m_i .

- x does not contain any $m \in M$. Then $x \in A + ek$.
- x contains any $m \in M$, which is not at the left of x . In this case it is possible to reduce x to zero with the rules in W_M .
- x contains one $m \in M$, and does contain that m at the left. Then $x \in m_1A \oplus \cdots \oplus m_sA$ and more specifically $x \in mA$.

Multiplication of two monomials a and b is as follows:

- if $a, b \in A + ek$ then $a \cdot b \in A + ek$ (apply reduction with the rules in W_e if necessary).
- if $a \in A + ek, a \neq 1$ and $b \in m_1A \oplus \cdots \oplus m_sA$ then $a \cdot b = 0$ (after reduction with W_M).
- if $a = 1 \in A + ek$ and $b \in m_1A \oplus \cdots \oplus m_sA$ then $a \cdot b = b$.
- if $a \in m_1A \oplus \cdots \oplus m_sA$ and $b \in A + ek$ then $a \cdot b \in m_1A \oplus \cdots \oplus m_sA$ (apply reduction with W_e if necessary).
- if $a, b \in m_1A \oplus \cdots \oplus m_sA$ then $a \cdot b = 0$ (after reduction with W_M).

□

Define $A_0 \subseteq A$ to be the set of monomials with constant 0. Consider the mapping from $F/\langle FW_MF \rangle$ to $G/\langle GW_MG, GW_eG \rangle$ which is formed by right-multiplication with e . This mapping is injective and the only elements that are mapped onto something in another residue class are the constants k that are mapped into ek , so $F/\langle FW_MF \rangle \cong (A_0 + ek) \oplus m_1A \oplus \cdots \oplus m_sA$ as A -modules and even as rings, with multiplication of two monomials a and b as follows:

- if $a, b \in A_0 + ek$ then $a \cdot b \in A_0 + ek$ (apply reduction with the rules in W_e if necessary).
- if $a \in A_0 + ek$ and $b \in m_1A \oplus \cdots \oplus m_sA$ then $a \cdot b = 0$ (after reduction with W_M).
- if $a \in m_1A \oplus \cdots \oplus m_sA$ and $b \in A_0 + ek$ then $a \cdot b \in m_1A \oplus \cdots \oplus m_sA$ (apply reduction with W_e if necessary).
- if $a, b \in m_1A \oplus \cdots \oplus m_sA$ then $a \cdot b = 0$ (after reduction with W_M).

Theorem 2.4. Let $R \subseteq A \subseteq F$ and $W \subseteq A^s \subseteq F$. Map these rules onto $F/\langle FW_MF \rangle$, by mapping A onto A and the i -th dimension of A^s onto m_iA . Map these relations into $G/\langle GW_MG \cup GW_eG \rangle$ by multiplying with e . Call the total set of relations $W^* \subseteq G/\langle GW_MG \cup GW_eG \rangle$. The Gröbner basis of $W^* \cup W_M \cup W_e$ contains the Gröbner basis of R and is finite if the Gröbner basis of R is finite. Furthermore the part of the quotient algebra starting with m_i gives a representation of the A -module with relations R and W .

Note that in the implementation it is not necessary to add W_M and W_e explicitly.

A basic set for modules

Here I will look which obstructions $(l, i, r; \lambda, j, \rho)$ need to be considered for obtaining a basic set of a combination $R \cup W \subseteq G / \langle GW_M G \cup GW_e G \rangle$, where $W \in A$ and $R \in m_1 A \oplus \dots \oplus m_s A$. I will use that either $r = 1$ or $\rho = 1$ (see lemma 2.1).

- Both $i, j \in W$. Without loss of generality take $r = 1 : ([, i, []; [], j, \rho)$, or $L(w_i) = L(w_j)\rho$. Note that this is a weak obstruction, unless it is a reduction of a module relation by another module relation.
- One of i and j is an element of W . Let w_i be the module relation.
 - $r = 1 : ([, i, []; \lambda, j, \rho)$ or $L(w_i) = \lambda L(g_j)\rho$. Note that this is a weak obstruction, unless it is a reduction of a module rule by a two-sided rule.
 - $\rho = 1 : ([, i, r; \lambda, j, [])$ or $L(w_i)r = \lambda L(g_j)$.
- Both $i, j \in R$. This will lead to the same obstructions as with the two-sided Gröbner basis. If the set of two-sided relations is already a two-sided Gröbner basis, this will lead to weak obstructions.

Since the whole Gröbner basis for the relations in G is needed, it might as well be calculated at the start and indeed doing so will make the algorithm faster in practice.

Calculating a (partial) quotient algebra

There are cases where it is not necessary to calculate the whole Gröbner basis to be able to produce the quotient algebra. When the quotient algebra of the partial Gröbner basis with obstructions is finite, it should be possible to construct the quotient algebra by checking the same relations (possibly in a slightly adapted form) as used in vector enumeration.

Chapter 3

Vector enumerator

3.1 Introduction

Steve Linton has written two articles about vector enumeration. The first is called “Constructing Matrix Representations of Finitely Presented Groups” [6] and the second is called “On vector enumeration” [7].

In this chapter a brief outline of the algorithm will be given. Proofs are omitted here.

3.2 Definitions

Let R be a basis for an ideal $I = \langle ARA \rangle$ in a free noncommutative algebra A . Let P be A/I . Let A^s be the free module of rank s over A . Let $W \subseteq A^s$ be a finite set of relations. It is sometimes possible to compute $(A/I)^s/\bar{W}$, where \bar{W} is the image of W in $(A/I)^s$.

3.3 Table

The information is stored in a table. The indices are taken from a set of available indices. For a row with index b the entries are

- p_b : the monomial of the row,
- d_b : true if p_b is a linear combination of smaller monomials (if d_b is true the entry is called *deleted*),
- r_b : reduction if reduction is possible,
- $v_{b,x}$: result of $p_b * x$.

The values of $v_{b,x}$ are entered as they become known. The table is started with the monomials $p_b(i)$, which denote the i module generators.

3.4 Adding rows to the table

When an equation is evaluated, the monomials are constructed by right-multiplication. For each intermediate value that cannot be reduced and is not already in the table, a new row in the table will be added. This means that all prefixes of a monomial (including the monomial itself) should either be reducible or in the table.

3.5 Applying the relators

The basic step consists of checking some equations, which are known to hold. Three types of equations are checked:

- For all rows in the table it is checked that $p_b \cdot w_i = 0$ (for $w_i \in W$) (comparable with an S-polynomial between a module and an algebra relation).
- For all module rules it is checked that they hold.
- And last $p_b x = p_b x$. This is an extra equation to calculate all $v_{b,n}$ that are \perp .

If the equation does not hold yet, then a new relation has been discovered. Such a relation is called a *coincidence*.

3.6 Pressing coincidences

If a linear relation is found among the undeleted members of the table it is possible to reduce the table, by substituting one of the monomials with the relation found for that monomial.

In case more than one coincidence is found at the same time, a stack is used.

3.7 Starting and finishing conditions

At the start s rows are created in the table. The monomials of those rows, $p_b(i)$, are the s module generators.

If at a certain moment all the relations are satisfied and the table is closed (all $v_{b,x}$ are unempty), the algorithm is finished. Note that the algorithm only finishes if the quotient algebra is finite.

3.8 Comparison

The strength of vector enumeration lies in the ability to find small modules in larger spaces. It might be worse when calculating larger quotient spaces, especially those for which there is a relatively small Gröbner basis.

Chapter 4

Making GBNP a GAP-package

4.1 Introduction

Several steps have to be taken in order to make GBNP a GAP package. In this chapter some of these actions will be described:

- supply documentation in: `GAPDoc`, `README`, `PackageInfo.g`
- rename functions: use a record and unique function names.
- structure the package, a part of which is to use `.gi` and `.gd` files, combined with the functions `DeclareGlobalFunction` and `InstallGlobalFunction`.
- look for better interworking with GAP.
- make some test cases (maybe from the examples) to be checked with `ReadTest`.
- verbose functions should use `Info` instead of `Print`.

For more information on the structure of a package, see Appendix A from the example package documentation, which is found at <http://www.gap-system.org/pkg/example/htm/chap00A.htm>.

4.2 Documentation

4.2.1 GAPDoc

GAPDoc is a GAP package which defines an abstract markup language for GAP-documentation using XML. The package also provides the tools needed to convert this documentation to the formats needed by GAP. More about GAPDoc can be found at the GAPDoc website:

<http://www.math.rwth-aachen.de/~Frank.Luebeck/GAPDoc/>

The new documentation for the GBNP package is based on the old documentation on the web, including some information from the files (functions and arguments). For each function intended to be called by the user, has a documentation in GAPDoc format has been made, using the comments from the files.

The list of files has been moved to an appendix and a new appendix has been made consisting of the examples. The examples have been altered a bit, so that they will produce less output if necessary and references have been added to other parts of the documentation. More information about this can be found in Section 4.2.4.

References to files must be changed to references to functions or sections of functions when possible.

4.2.2 GAPDoc:Include

It is possible in GAPDoc to split documentation over multiple files. In particular it is possible to include function descriptions in GAPDoc markup of information from before the functions in the source-code. A piece of GAPDoc is delimited by “`### <#GAPDoc>`” and “`### </GAPDoc>`” (where “`###`” are the comment characters). Such a piece can be included with `<Include Label="Name">`. Note that this is not a valid XML element, only something to be preprocessed by GAPDoc. A small example can be found in the next section.

The list of all files which are checked for pieces of GAPDoc markup is found in `make_doc.g`.

4.2.3 GAPDoc:Functions

The descriptions of the functions (including descriptions of arguments and return values) are put in the source (`.gi`-file) as much as possible. This makes it easier to adjust it, when a function is changed. GAPDoc Markup is added to the comments before a function so that they can be included in the package documentation.

Including files in GAPDoc is described in Section 4.2.2. Only the files listed in `make_doc.g` will be checked for GAPDoc-parts (see also Section 4.2.6). Below a small example will be given from `grobner.gi`:

```
### <#GAPDoc Label="Grobner">
### <ManSection Label="Grobner">
### <Func Name="Grobner" Comm="Buchberger's algorithm with normalform"
### Arg="KI" />
###
### <Returns>
### G, a Gröbner Basis (if found...the general problem is unsolvable).
### </Returns>
###
### <Description>
### For a list of noncommutative polynomials <A>KI</A> this function will use
### Buchberger's algorithm with normalform to find a Grobner Basis
### <C>G</C> (if possible, the general problem is unsolvable).
### </Description>
### </ManSection>
### <#/GAPDoc>
```

The outermost `#GAPDoc`-tags are for the preprocessor to know what to include. The `ManSection` tag is explained in the GAPDoc manual, so I will only explain it briefly here. It is a subsection used for explaining functions, variables, methods, inforevels, etc. The `Func` element describes which function the `ManSection` is about and what the name and arguments of the function are (and has an optional comment, which should describe the function in one sentence). The `Returns` element gives information about the return value(s) of the function. The `Description` element (which mentions the arguments again, because only the names were mentioned before) should describe in some more words what the function does. Several other functions could be mentioned here (like `Preprocess`, which is mentioned in `EPFinCheck`). For more informations about references see 4.2.5.

Some other GAPDoc tags that are used are `<A>..` (function argument) and `<C>..</C>` (code). More information can be found in the GAPDoc manual.

This example can be included in a section with Gröbner functions as follows:

```
<Section Label="grobner">
  <Heading>Gröbner functions, standard variant</Heading>
  <#Include Label="Grobner">
  <#Include Label="SGrobner">
</Section>
```

4.2.4 GAPDoc:Examples

Introduction

Including examples in the documentation makes it easier to view them when viewing a link-capable version of the documentation and allows to include links to relevant sections of the documentation functions that are used.

Decrease the output

Some examples produce large sums of output. It is impossible to include all of this in the documentation. Most of this large output is not even necessary and can be hidden with `;;`, or by setting a lower `InfoLevel` (see 4.6).

Sometimes only a part of the result suffices to demonstrate something and therefore only a part will be shown (like in Example 8, an example of the trace variant).

References from examples

A lot of references can be added from the examples to provide a better interaction. It is possible to add a link for most functions in the text and sometimes a reference to a section. For references in GAPDoc, see Section 4.2.5.

Changes

Apart from reducing output and adding text and references, some changes have been made to the examples. Some comparable functions have been added or the examples have been extended a bit (like in Example 6, where it is shown how it is easier to input the relations with `GP2NPList`). Examples 16 and 17 have been combined, because there was a large overlap.

Auto generation and testing: conversion details

The files that are included are mostly in xml-format. These xml-files are generated from `.g`-files by running GAP and then a unix program: `sed`.

First GAP is run with the command (where `#{1}` is the name of the file without `.g`):

```
{ echo "LogTo(\"#{1}.txt\");"; cat "#{1}.g"; } |gap
```

This will produce a txt file which can be used with `ReadTest`. An XML file can be created with a few regular expressions:

```
s,^gap> $,,
s,<L>,<Listing><!\[CDATA\[,
s,</L>,\]\]></Listing>,
s,^gap> # ,,
```

The first line will empty all lines which were empty. The second and third line function as an abbreviation for `<L>` and `</L>` tags which can be used as a small tag around GAP code listings. (When including a whole example in the documentation this tag is used around all pieces of GAP-code. The non-GAP-code will be after a `'# '`). The fourth line removes `gap > #` from before the lines that are actually XML code.

In the generation of the examples without the comments the command `grep -v '^#'` is used.

Auto generation and testing: an example file

```
# <#GAPDoc Label="Example00">
# <Section Label="Example00"><Heading>Example 0</Heading>
# This example will demonstrate the loading of <Package>GBNP</Package>.
# <P/>
```

```
# Require the loading of the package and set the standard info level to 2 and
# the time infolevel to 1 (for more information about the info level, see
# chapter <Ref Chap="Info"/>).
```

```
# <L>
RequirePackage("GBNP");
SetInfoLevel(InfoGBNP,2);
SetInfoLevel(InfoGBNPTime,1);
# </L>
```

```
# </Section>
# <#/GAPDoc>
```

<Include Label="Example00"> could then be used to include this example in a GAPDoc document after the filename of the example has been added to `make_doc.g`.

Auto generation and testing: how to make different example formats

To make the different formats of the examples go to the directory `doc/examples/`. There is a `GNUmakefile` for autogeneration of the necessary files. The original files have extension `.g`. The output from GAP is in the files with extension `.txt` and the files that will be included have extension `.xml`. There is also an `\examples` directory where the examples without comments can be found (for cut and paste). These examples can be generated there with `make example<n>.g` where `<n>` is the number of the example.

Auto generation and testing: how to make tests

For each example it is possible to make a test in `test/` by going there and typing `make name.test`. To check the tests type `make all.txt` (if necessary use `touch all.g` to make sure this file is newer). If all differences are explainable it is possible to type `make *.test` to update the test files. It is possible to do something with the `GAPStones` which are an indicator for speed, but this is not done yet (for more info on `GAPStones`, see also (Reference: Test Files)).

4.2.5 GAPDoc:References

In GAPDoc it is possible to add references, which can be converted to produce clickable links. This can be very useful, especially in examples. Most references are to functions, sections of this manual or sections of another GAP manual.

GAPDoc references

The standard reference in GAPDoc will look like `<Ref What="Name">`, where `What` is the type of the reference (to a function (`Func`) or maybe to a chapter (`Chap`) and `Name` is the text of the label which is given as an attribute in the item that is referred to (`<Func Label="example" .../>`).

More can be found about this in the GAPDoc documentation on `<Ref>`.

When entering documentation, it might be useful to make macros for function and section references, since they occur often.

Function references

In a function reference (`<Ref Func="Name" Style="Text"/>`) it might be useful to add the `Style="Text"` attribute which will also print the name of the function. The `Style="Number"` attribute should only print the number.

Section references

When using references to a section (`<Ref Type="Sect"/>`) it is advised not to use the option `Style="Text"` because this might give errors.

References to other GAP manuals

A references to another part of the GAP manual (`<Ref BookName="<book>" Label="<topic>"/>`) will insert a link to the place which would be found with `?<book>: <topic>` from inside GAP. If it is not possible to use external links, just `<book>: <topic>` will be printed instead. An example would be `<Ref BookName="GapDoc" Label="Ref"/>`.

Bibliography and URLs

It is possible to add URLs with `<URL ...>`. It is possible to add references to the bibliography with `<Cite Key="Name"/>`. I have found no way to include references without citing them (like the \LaTeX -command `nocite`).

To include the url in the bibliography I used a \LaTeX -package called `urlbst` (it can be obtained from CTAN:biblio/bibtex/contrib/urlbst/) which makes it possible to create bibtex styles which support URLs. I copied the file `plainurl.bst` from the package to the documentation directory and can now use `Style="plainurl"` in the Bibliography tag, which will lead to the \LaTeX -command `\bibliographystyle{plainurl}`.

4.2.6 Producing the GAPDoc documentation

The examples have to be run to generate the XML files which can be included with GAPDoc. See also 4.2.4. There is a `make` target that runs the examples and makes the documentation. This target is `doc`.

The documentation can be generated from inside GAP with the GAPDoc-package by use of a script (there is also a chapter about this in the GAPDoc-documentation). An annotated version of this script can also be found in the main directory of the package and is named `make_doc.g`. Reading this document into GAP should be enough to update the documentation.

Whenever a new version of the documentation is produced we can run this script to create the new files and then later package them.

4.2.7 README

Each package needs a `README` file, including installation instructions and names of the package authors and their email-addresses. This file is created from the original text documentation. Installation instructions were added. The address of the authors has been added at the bottom of the file.

4.2.8 PackageInfo.g

In this file a lot of info about the package and maintainers can be found.

Some things that still need to be checked:

- The current `Status` here is “dev”, which might be changed later.
- The postal address for both A.M. Cohen and D.A.H. Gijssbers is set to RIACA.
- All URLs should be checked. The start is `http://www.win.tue.nl/~amc/pub/grobner/`, but not all files will actually be there.
- The (optional) title `PackageDoc.LongTitle` for use with `?books`, has been set to ‘Non-commutative Gröbner bases’.

- `BannerString` can be changed to make a banner other than the standard banner. The standard banner is:

```
-----
Loading  GBNP 0.9 (Non-commutative Gröbner basis)
by A.M. Cohen (http://www.win.tue.nl/~amc) and
   D.A.H. Gijssbers (D.A.H.Gijssbers@tue.nl).
-----
```

- `TestFile` contains tests (edited examples). It is best to test a lot of functions here, but to use only rather small examples. If there are test that take a long time (several minutes or more), those should be run locally at Eindhoven. Other tests can be run if people want to run the complete set of GAP package tests.

The only example that takes more than a few seconds to run is Example 10 (about 3 minutes).

- The easiest archive format seems to be `.tar.gz`. The other formats can be generated from that. There is a `Makefile` in the main package directory which will create the total archive and the documentation archive with the command `make` (they can be remade by first using `make clean`). This `Makefile` assumes that `gap` is in the path (for remaking the documentation).

4.3 Splitting declaration and implementation

In the `.gd` file the functions are declared and in the `.gi` files the functions will be implemented. More information about this can be found in the the GAP Programmers tutorial (`?Prg Tutorial:Declaration and Implementation Part`) and in the extending reference manual (`?Extending:The Files of a GAP Package`).

This comes down to the use of `DeclareGlobalFunction("FuncName");` and in the implementation:

```
InstallGlobalFunction(
FuncName,function(args)
...
end);
```

The `.gd` files will be read from `init.g` and the `.gi` files from `read.g`, with the commands `ReadPackage("file.gd");` or `ReadPackage("file.gi");`. In the last one the order does matter.

Note that GAP used `ReadPkg` instead of `ReadPackage` before version 4.4.

4.4 Function names

4.4.1 GBNP-record

Global function names should be unique. There are two issues: people can overwrite a function by mistake, which leads to bugs or if they would know about the functions they could be restricted because the name they want to use for a function of their own exists already in a package.

Since GBNP is not always loaded it is not a problem to use a lot of short function names, but it is useful to extend the names to something a user wouldn't choose. For example an arithmetic function with a short name like `Lt` had better be named `LtNP`, to allow the existence of another `Lt` procedure.

Local function names can be put in a record (`GBNP.*`) which protects them from being overwritten by accident, while they are still not taking up a large part of the namespace. The record `GBNP` is protected by declaring it as a global variable with `DeclareGlobalVariable` and `InstallVariable`. More information about this can be found at (`Prg Tutorial: DeclareGlobalVariable`). Note that this change also allows the declaration of local functions in any order.

These changes have mostly been done. The function `NondivMonsByLevel` has been put in the record, and the functions `StrongNormalForm(Trace)NP` have not, because they might be useful to solve the word problem. ?

4.4.2 Global Variables

It is not good to have global variables like `F`, `A` and `g` as originally used in `npformat.g`. The functions `NP2GPList` and `NP2GP` have been rewritten to include an argument `A`, the algebra. The field can be acquired with the function `LeftActingDomain` and the generators can be acquired with one of the functions `GeneratorsOfAlgebra` and `GeneratorsOfAlgebraWithOne`.

If it is not possible to do without a global variable it can be kept in the `GBNP` record or it has to be declared with `DeclareGlobalVariable`.

The variables in the `GBNP` record are combined into a single options record and there are some functions, like `GBNP.GetOptions()`, which can retrieve the options.

The advantage of giving the values as an argument of a function is that it is easier in successive calculations or when running something from a GAP break loop. The disadvantage is that this variable has to be entered every time by the user and given to a lot of internal functions.

4.5 Better interworking

There now are a few examples of entering relations from an associative free algebra created with the GAP-function `AssociativeFreeAlgebraWithOne`. Furthermore it is possible to display generators of NP-polynomials like they were displayed in the algebra with `GBNP.ConfigPrint`.

It might be possible to add other ways to allow better interworking. XXX suggestions.

4.6 Verbose Functions

Some functions print a lot of information. The `Print` statements have been changed to `Info` statements. This command enables the user to choose how much information will be printed. There is a new chapter in the documentation about this. The `Info` command has two extra parameters, the first is the `InfoClass` (which would be `InfoGBNP` or `InfoGBNPTime`) and the second is the level, from which the information will be evaluated and printed. Note that an `Info` command always prints an extra newline at the end. For more information see the `Info` chapter in the documentation of `GBNP`.

The extra newline in `PrintNPListTrace` has been removed so the output of this function looks more like that of `PrintNPList`. The function `PrintTraceList` still prints a newline after the polynomials but no longer also after the last one.

4.7 Tests

The only tests at the moment are the examples. They can be automatically generated, except for the file `all.g`, in which the files to test are added by hand. For more information about the autogeneration of test files see Section 4.2.4.

4.8 Printing symbols

There are some new ways to print symbols. These different options can be set using the function `GBNP.ConfigPrint`, which changes the function `GBNP.Transletter`.

It is possible to use the syntax to name symbols the same way as when constructing a free algebra (Reference: `FreeAssociativeAlgebra`). It is also possible to give an algebra as an argument and the function will use the names from that algebra (it gets them from the family with `ElementsFamily(FamilyObj(a))!.names`). This is used in Example 21 (?GBNP: Example 21).

This function is described in detail in the package documentation.

Chapter 5

Some improvements in the implementation

5.1 Introduction

First I used profiling (see also Section A.2) to check which functions took the most time, to concentrate on improving those. I noticed that a lot of time was spent in the `GBNP.Occur` function. This is the function that is used to check whether a monomial is contained in another monomial. I have tried to reduce the number of calls of this function (see Section 5.2) and to speed it up.

In some cases it is enough to check on the right, which is what `GBNP.RightOccur` does. This function can be done quicker with a tree structure (see Section 5.3).

This gave such an improvement that I tried to adapt this data structure so that it could be used in more cases. This will be described in the Sections 5.4 and 5.5.

When using very large input some other problems arise, such as the time it takes to do operations on lists. This will be discussed in Section 5.7, together with some suggestions.

5.2 Occur

There are some possibilities to speed up the function `GBNP.Occur`, the fastest of which is the tree structure which will be described in Section 5.4. There are some other improvements, but they cannot be combined with the improvements from the tree structure. But they can be used where the latter cannot: in parts IIIc and IIId of the loop, where G and *todo* are reduced with the added polynomial.

The Number of calls to `Occur` can be further reduced by using cheap checks in advance. It is possible to check that all the generators of the substring are in the word. This can be checked quickly by use of boolean lists, see Reference: Boolean Lists (an efficient way to store and calculate with lists of booleans). A boolean lists will have to be stored for each leading term, but the relative amount of space needed is small (about $\frac{1}{8}$ th the number of generators in bytes per leading term).

5.3 RightOccur

For some of the calls to `GBNP.Occur` it is only necessary to check whether a leading term occurs at the end of the monomial. These calls can be replaced by calls to `GBNP.RightOccur`, which only does that and hence is quicker.

A place where this is very useful is in `GBNP.NondivMons`, where the basis of the quotient algebra is calculated. A monomial is in the basis of the quotient algebra if it cannot be reduced by the Gröbner basis. A way to generate all basis elements is by starting with 1 as a candidate and

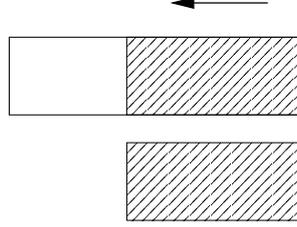


Figure 5.1: RightOccur: does a monomial (above) contain a monomial from a list (below) at the end? The arrow indicates the direction in which the search is done.

obtaining other candidate basis elements by right-multiplying a basis element with a generator of the algebra and then checking whether it can be reduced.

Lemma 5.1. *For checking reduction in `GNBP.NondivMons` it is enough to check the tail (or right part) of a candidate basis element.*

Proof. Let w be a basis element of monomial basis of the quotient algebra and a a generator of the algebra. Note that wa is a basis element of the quotient algebra iff it does not contain a leading term of the Gröbner basis. Since w is a basis element of the quotient algebra, it does not contain a leading element of the Gröbner basis and therefore the only place where wa can contain a leading term of the Gröbner basis is on the right. \square

Instead of calling `GNBP.Occur` it is possible to call a function that only checks whether the tail (or right part) matches a leading term of the Gröbner basis. The function `GNBP.RightOccur` does that.

5.4 RightOccurInLst, RightOccur tree-structure

It is also possible to speed up repetitive calls of `GNBP.RightOccur` by building a tree structure (it could also be seen as an implementation of a deterministic finite state automaton). How this can be done will be described in this paragraph. This technique is especially useful if the Gröbner basis is large.

5.4.1 Recursive definition

Arguments for the function f will be a list of numbered monomials G and a monomial p . The function should find one of the monomials in G that is at the end of p and then return the corresponding number. If such a monomial can not be found in G then it should return 0.

The data structure used for G makes testing if a monomial of length 0 is contained easy (so $f.G.\perp$ is considered to be known.)

It is clear that $f.\emptyset.p = 0$ and that if $(\perp, i) \in G$ for some i then $f.G.p = i$. Now suppose $(\perp, i) \notin G$. Then $f.G.\perp = 0$ and if a is the last symbol of p ($p = p' \dashv a$) then $f.G.p = f.G.(p' \dashv a) = f.G'_a.p'$, where $G'_a = \{(m, i) \mid (m \dashv a, i) \in G\}$. Or in formulas:

$$f.\emptyset.p = 0$$

$$f.G.p = \begin{cases} i, & \text{if } \exists_i(\perp, i) \in G \\ 0, & \text{if } \neg \exists_i(\perp, i) \in G \wedge p = \perp \\ f.G'_a.p', & \text{if } \neg \exists_i(\perp, i) \in G \wedge p = p' \dashv a \end{cases} \quad (5.1)$$

It is possible to create a tree structure so that all G'_a can be obtained from it. Nodes with $(\perp, i) \in G$ will be end nodes with the number of the matching leading term from G (note that the

empty tree will have to be a special case). Any other node will have outgoing branches for each a which leads to a non-empty G'_a .

The empty tree can be represented by one node, with the number 0 or by a node with an empty list of branches (the latter is more useful when constructing trees).

5.4.2 An example

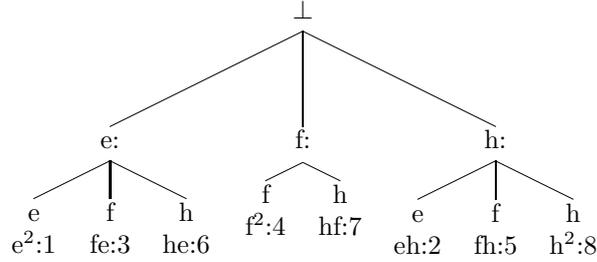
Consider the following Gröbner basis:

$$e^2, eh + e, fe - ef + h, f^2, fh - f, he - eh - 2e, hf - fh + 2f, h^2 - 2ef + h$$

of which the leading terms are:

$$e^2(1), eh(2), fe(3), f^2(4), fh(5), he(6), hf(7), h^2(8).$$

Its tree is:



5.4.3 Creating a tree

A tree can be created by starting with the empty tree and then adding monomials (leading terms) and their numbers. In this section we give a brief indication of how this works.

Adding a monomial can be done by extending the tree recursively. Suppose the monomial m with number n is added to G and define $G^* = G \cup \{(m, n)\}$.

Recall the relations from 5.1:

$$f.G \cup \{(m, n)\}.p = \begin{cases} i, & \text{if } \exists_i(\perp, i) \in G \cup \{(m, n)\} \\ 0, & \text{if } \neg \exists_i(\perp, i) \in G \cup \{(m, n)\} \wedge p = \perp \\ f.(G \cup \{(m', n)\})'_a.p', & \text{if } \neg \exists_i(\perp, i) \in G \cup \{(m, n)\} \wedge p = p' \dashv a \end{cases} \quad (5.2)$$

After splitting of m this results in:

$$f.G \cup \{(m, n)\}.p = \begin{cases} i, & \text{if } \exists_i(\perp, i) \in G \\ 0, & \text{if } \neg \exists_i(\perp, i) \in G \wedge p = \perp \wedge m \neq \perp \\ n, & \text{if } \neg \exists_i(\perp, i) \in G \wedge p = \perp \wedge m = \perp \\ f.(G \cup \{(m', n)\})'_a.p', & \text{if } \neg \exists_i(\perp, i) \in G \cup \{(m, n)\} \wedge p = p' \dashv a \end{cases} \quad (5.3)$$

This can be obtained by following the path for m (along other words with the same substring), then extending it, when there are no other words with the same substring and then adding the value, n .

5.4.4 LeftOccur-trees

It is possible to do the same from the left as from the right. This is very useful if there are prefix rules, which can only apply from the left. A *LeftOccur-tree* is a *RightOccur* tree, with the exception that the words are entered in reverse.

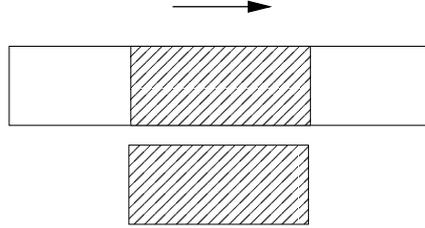


Figure 5.2: OccurInLst: does a monomial (above) contain a monomial from a list (below)?

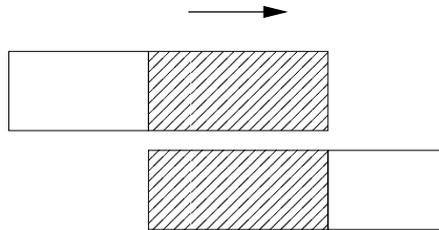


Figure 5.3: left obstruction

5.5 Other uses for OccurTrees

5.5.1 Introduction

So far this test has only been applied at the right. But with a little change in the datastructure it is possible to apply it from any startposition in a monomial (see 5.5.2).

Furthermore it is possible to adapt the functions a bit to speed up searches for obstructions aswell (here only an overlap is needed).

To keep the data structure up to date, it is necessary to handle deletions, insertions and sorting of G and *todo*. This can be achieved by the use of two arrays (as introduced in 5.5.5).

5.5.2 Using trees in OccurInLst

To check whether a monomial from the list R occurs in a monomial m , it is enough to check whether a monomial from R occurs from each position p . Furthermore only the first match is returned, so it is enough to return the check the first positions until a match has been found.

The shortest (smallest) monomial will always match first and will be the one returned.

5.5.3 Obstruction searching

To find all left obstructions between a monomial and a list of monomials, one must find all monomials where the end of the first monomial is the beginning of the other monomial. When given an ending part of the first monomial it is possible to use the tree-structure to look which monomials in the list start with that. The same can be done for right obstructions.

More formal: remember the definition of f from 5.1. We will define a recursive definition g . The search now is for all matchings. If G is empty then there will be no matching monomials, or $g.\emptyset.p = \emptyset$. If $p = \perp$ then for all matching monomials i we must have $(m, i) \in G$ for certain monomial m . The reason why the tree structure can be used unchanged is that the recursion is the same. For left (right) obstructions define $G'_a = \{(m, i) | (a \vdash m, i) \in G\}$ ($G'_a = \{(m, i) | (m \dashv a, i) \in G\}$)

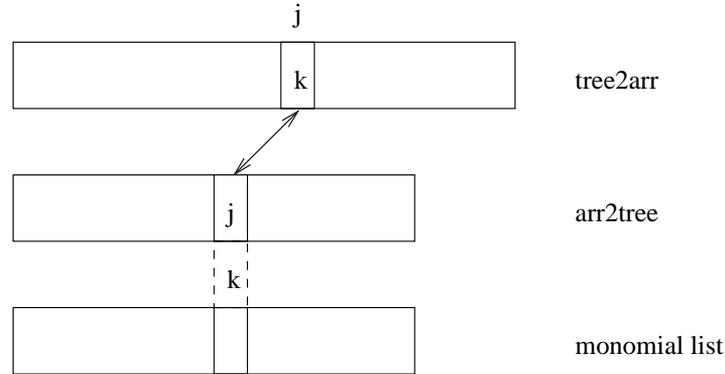


Figure 5.4: arr2tree and tree2arr

and $p = a \vdash p'_a$ ($p = p'_a \dashv a$).

$$\begin{aligned}
 g.\emptyset.p &= \emptyset \\
 g.G.p &= \begin{cases} \{i \mid \exists_m(m, i) \in G\}, & \text{if } p = \perp \\ g.G'_a.p', & \text{if } p = a \vdash p'_a \end{cases} \quad (5.4)
 \end{aligned}$$

5.5.4 Removing elements and sorting arrays

During the Gröbner-loop the list G and $todo$ are changed: elements are inserted and deleted and occasionally the whole list is sorted. This would make the calculated tree useless, and a new tree would have to be calculated. It is interesting to look for techniques that allow the numbers in the tree to be the same. The numbers in the end nodes will no longer correspond to the place in the list, but will first have to be translated. This is achieved by using two extra arrays (see also Section 5.5.5).

It is also possible to use pointers instead of a number. When the array is sorted and the numbering changed, it is then no longer necessary to update the tree structure. Furthermore it is possible to access the polynomials more directly. The disadvantages are that it involves a lot of recoding and possibly slows down other procedures. Because of the amount of recoding this has not been done.

5.5.5 Extra arrays tree2arr and arr2tree

At an end node in a tree there will be a unique number, say j . The position of the corresponding monomial in the list of monomials can be found in the array `tree2arr`. The number used in the tree for a monomial k from the list can be found at position k in the array `arr2tree`.

After changing the order of the list of monomials or after inserting or deleting an element, the information in the array `tree2arr` has become partially invalid. This can be fixed by setting `tree2arr[arr2tree[i]]=i` for all positions i where the monomials have been changed.

The numbers in the tree must all be unique. It is possible to pick a new number for every new monomial, but this will cause the array `tree2arr` to become very large. Instead a linked list is used to keep track of which elements are not used, to be able to reuse them whenever possible.

5.6 Other improvements

5.6.1 Boolean lists

A boolean list is an efficient representation for a small set (like the set of generators of the algebra and maybe the module if there is any). It is possible to reduce the number of calls to `Occur` by

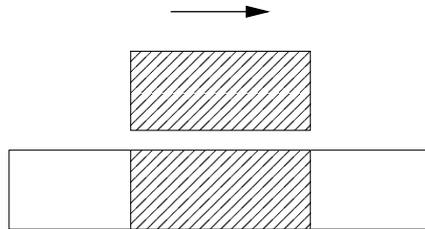


Figure 5.5: No occur trees can be used easily here.

checking whether a smaller word has symbols which all occur in the larger word before the call to `Occur`. Since this check is more efficient than the check in `Occur` this can save quite some time if not all monomials that are checked contain all generators (this is not very rare when there are more generators, if there are only 2 generators then this will rarely be an improvement).

This is however slower than using occur trees and therefore remains only in a smaller part of the code. Especially in some places, where it is hard to implement occur trees. For example when it is checked if a lot of polynomials can be reduced with one polynomial.

5.6.2 Using a merge sort when adding

An addition was previously done by concatenating the lists and then calling `CleanNP` (which sorts the lists). If the polynomials are already sorted then it is possible to use a faster kind of sorting: merge sort. This has been implemented.

5.7 Further suggestions

5.7.1 Data structures

If the size of the input is very large (more than 10000 relations in the Gröbner basis). Other parts of code start to cost time. Some of these are not completely optimized yet, while others come from the inefficient handling of large lists. It might be useful to implement a data-structure which allows for easy smallest element extraction and easy insertion and replacement.

Two possible candidates are Binary Search Trees and 3-Heaps. These will not be described in detail here. These should be considered only if sorting and list assignment are starting to take a large part of the algorithm's time.

It might also be useful to use records as elements of the array instead of polynomials. The advantage would be that the object does not change (only the polynomial member does) which makes it easier to keep track of polynomials in an occurtree. The disadvantage is that this may give a small overhead.

5.7.2 PositionSublist

The GAP function `PositionSublist` can be faster in finding sublists but a lot of time is spent initialising. It is possible to write code that does the initialisation only once for a leading term. This can be useful when checking whether a (small) list is occurring in several lists. This has not been implemented because most of such checks have been removed with the introduction of `OccurTrees`.

5.7.3 Other places to speed up

In some places it is not possible to use occur trees. If these start to take a large part of the time, than it might be useful to speed these up in another way. This will however complicate the

algorithm further. A possibility might be to check for several monomials instead of one (so that a small occur tree can be used). This can be seen as a kind of prediction. If these results maintain valid then they can be used.

5.7.4 Compiling

It is possible to compile functions in GAP. Compiling GAP-code has not made the code much faster. It might still be useful to check whether rewriting the functions `LtNP` and `LookUpOccurTree*` in c and leaving out unnecessary checks gives an improvement.

5.7.5 Partial prefix basis

It might be possible to stop the prefix and two-sided variant of the Gröbner basis algorithm earlier if only a module is wanted. The basic idea is that if at a certain moment only a finite number of prefixes cannot be reduced then the current relations are sufficient information to construct the module.

5.7.6 Combining variants

The code will be easier to maintain if it does not consist of several versions of the same algorithm which are slightly modified to be another version. It might be easier to maintain if variants are combined. This also allows a combination of variants (a trace for a truncated basis for example). But the resulting single procedure may be harder to understand.

Some work has been done to some combined optimized procedures, instead of optimising all different variants. This is not finished and more work might be spend on this.

5.7.7 Optimisations where needed

Not all code is optimised. Mostly the places where a lot of time was spent, were optimised. This might mean that for some cases the algorithm is not as fast as it could be. It might be useful to consider profiling and specific optimisations if the total runtime is very long.

Chapter 6

Application: Lie algebras

6.1 Introduction

One of the tasks of my internship was, if possible, to test the package on the search for finite-dimensional associative algebras, of which the corresponding Lie algebra contains a given Lie algebra. This chapter is about that part. First I will tell some things about Lie algebras. Then I will tell some things I did and explain some of the results.

6.2 Introduction to Lie algebras

In this section a brief introduction to Lie algebras will be given. A good reference to Lie algebras is [4].

In this chapter the Lie algebra product of x and y will be denoted $[x, y]$, the bracket product.

Definition 6.1. A Lie algebra is a non-associative (not necessarily associative) algebra if the following relations hold:

$$[x, x] = 0 \tag{6.1}$$

$$[x, y] = -[y, x] \tag{6.2}$$

$$[[x, y], z] + [[y, z], x] + [[z, x], y] = 0 \tag{6.3}$$

The second relation follows from the first ($0 = [x + y, x + y] = [x, x] + [x, y] + [y, x] + [y, y] = [x, y] + [y, x]$), but the first only follows from the second if the characteristic $\neq 2$.

The third relation is called the *Jacobi identity*.

Definition 6.2. It is possible to create a Lie algebra from an associative algebra A (for example from an algebra of square matrices) by defining the Lie product as follows:

$$[x, y] = xy - yx$$

The Lie algebra obtained this way is called the corresponding Lie algebra of A (notation $L(A)$).

Extremal elements

Definition 6.3. Let L be a Lie algebra. An $x \in \mathcal{L}$ is called **extremal** if $\text{ad}_x^2(\mathcal{L}) = \langle x \rangle$ (where $\text{ad}_x^2(\mathcal{L}) = [x, [x, \mathcal{L}]]$).

This means that for each y the element $[x, [x, y]]$ can be written as a scalar multiple of x : $[x, [x, y]] = 2g(x, y)x$. If written out in the Lie algebra of an associative \mathcal{A} the equation becomes:

$$x(xy - yx) - (xy - yx)x = 2g(x, y)x$$

or

$$x^2y - 2xyx + yx^2 = 2g(x, y)x \quad (6.4)$$

If $x^2 = 0$ this can be reduced to

$$-2xyx = 2g(x, y)x. \quad (6.5)$$

Enveloping algebra of a finite dimensional Lie algebra

Definition 6.4. *Given a finite dimensional Lie algebra g over a field k , the enveloping algebra, denoted by $U(g)$, is defined to be the associative k -algebra generated by x_1, \dots, x_n subject to the relations:*

$$x_jx_i - x_ix_j = [x_j, x_i], 1 \leq i < j \leq n$$

These relations form a Gröbner basis for the (infinite) universal enveloping algebra.

6.3 Construction

6.3.1 Introduction

The object is to find a finite dimensional (quotient) algebra of which the corresponding Lie algebra contains a given Lie algebra. This is done in particular for split semi-simple Lie algebras.

First the simple Lie algebra is generated in GAP with the function `SimpleLieAlgebra`. Over the rationals these give the Lie algebra with respect to a Chevalley basis (first the simple positive root elements, then the other positive root elements, then the corresponding negative root elements and as last the basis of the Cartan subalgebra). For the remainder of this chapter they will be called (e_i, f_i, h_i) (note that there are more e_i and f_i than h_i).

Then the relations of the universal enveloping Lie algebra are calculated: for all generators x_i and x_j of L with $x_i < x_j$ add the relation $x_ix_j - x_jx_i = [x_i, x_j] \Leftrightarrow x_jx_i - x_ix_j + [x_i, x_j]$.

Then one or more relations are added to the relations of the universal enveloping Lie algebra to try to make it finite and the Gröbner basis and quotient algebra are calculated.

We shall look for representations where extremal elements act quadratically, that is satisfy $x^2 = 0$.

Some of these representations of simple Lie algebras where the root elements are quadratic are described in [9]. Note that the results from that paper apply in the cases where the characteristic is not equal to two. For some Lie algebras characteristic 3 is a special case (as in the article).

Below it will be shown that any such representation can be found by adding extra relations and that if a calculation results in a vector space that does not contain L then no such vector space exists.

6.3.2 Construction

Let L be a Lie algebra. Let $\rho : L \rightarrow \underline{\mathfrak{gl}}(V)$ s.t. $\rho(x)^2 = 0$ for all $x \in E$, where E is the set of extremal elements in L . Let i be the Lie algebra embedding mapping L into $U(L)$ the universal enveloping algebra of L . Let $\tilde{\rho}$ be an associative algebra homomorphism from $U(L)$ to $\underline{\mathfrak{gl}}(V)$.

Lemma 6.1. *Suppose $\rho(x)^2 = 0$. Then $\rho((ix)^2) = 0$.*

Proof. If $\rho(x)^2 = 0$ then also $\tilde{\rho}(ix)^2 = 0$. Since ρ is an algebra homomorphism this means that $\tilde{\rho}((ix)^2) = 0$. \square

This means that a representation of a simple Lie algebra where the (extremal) root elements are quadratic can be formed in the following way: Add the relations $x^2 = 0$, to those of the universal Lie algebra and calculate the quotient algebra of the Gröbner basis. V can then be the quotient algebra and the transformations can be the left multiplication with the images of the Lie algebra generators and any other example is a quotient of this one.

6.3.3 Representations where only some root elements are quadratic

In principle $x^2 = 0$ for $x \in E$ gives infinitely many relations. However in practice it suffices to take only a few of these. In particular in A_1 , it follows from the relations $\{G(1) = fe - ef + h, G(2) = he - eh - 2e, G(3) = hf - fh + 2f\}$ and the additional relation $G(4) = e^2$ that f^2 also is equal to zero if the characteristic is not 2 or 3. In other words: if the positive root element is quadratic then so is the corresponding negative root element. The trace of the computation verifying this is:

$$\begin{aligned}
& - 1/24fG(1)ef^2 - 1/12fG(1)f - 1/24feG(1)f^2 - 1/12fG(1)hf + \\
& \quad 1/24f^2eG(1)f + 1/12G(1)hf^2 + 1/24G(1)ef^3 - 1/24fG(1)ef^2 + \\
& 1/24eG(1)f^3 - 1/24f^3eG(1) - 1/24feG(1)f^2 + 1/24f^2G(1)ef + \\
& 1/12G(1)f^2 - 1/24feG(1)f^2 + 1/24f^2G(1)ef + 1/12f^2G(1) + \\
& 1/24f^2eG(1)f - 1/12fG(1)f + 1/24f^2G(1)ef - 1/12fG(1)hf - 1/6G(1)f^2 - \\
& 1/24fG(1)ef^2 + 1/6fG(1)f + 1/12f^2G(1)h - 1/24f^3G(1)e + 1/24f^2eG(1)f \\
& - 1/24f^2G(2)f + 1/24f^3G(2) - 1/24f^2G(2)f - 1/24f^2G(2)f + 1/24fG(\\
& \quad 2)f^2 - 1/24G(2)f^3 + 1/24fG(2)f^2 + 1/24fG(2)f^2 + 1/4G(3)f + \\
& 1/12fhG(3) - 1/12hG(3)f + 1/12feG(3)f - 1/12eG(3)f^2 - 1/12f^2eG(3) + \\
& 1/12fG(3)h - 1/12G(3)hf + 1/12feG(3)f - 1/4fG(3) + 1/2fG(3) + \\
& 1/24f^2G(4)f^2 + 1/24G(4)f^4 \\
& - 1/24fG(4)f^3 + 1/24f^2G(4)f^2 - 1/24f^3G(4)f - 1/24f^3G(4)f + \\
& \quad 1/24f^2G(4)f^2 - 1/24fG(4)f^3 - 1/24f^3G(4)f - 1/24fG(4)f^3 + \\
& 1/24f^2G(4)f^2 + 1/24f^2G(4)f^2 + 1/24f^2G(4)f^2 - 1/24f^3G(4)f - \\
& 1/24fG(4)f^3 + 1/24f^4G(4)
\end{aligned}$$

When in A_2 the equality $x^2 = 0$ holds for one of the positive simple root element, then it also holds for the other positive simple root element, given that the characteristic is not 2 or 3 (there is a 6 in the trace output).

This is also the case for B_2 and G_2 for one of the root elements. This means that if adding the relation e_i^2 for each positive simple root element is the same as adding the relation for a single long root element. If there is a double or triple line in the Dynkin diagram then it might be possible to get a larger quotient algebra by choosing only one root. In G_2 this is even necessary, since adding the relation e_i^2 for all positive simple root elements will give a trivial quotient algebra.

6.3.4 GF(2)/GF(3)

Characteristic 2

In characteristic 2 adding these relations does not give the desired result. It is however possible to use the relation $e_i * f_i * e_i = e_i$ instead.

Characteristic 3

In characteristic 3 only some small changes occur. A_1 , B_2 (1) and C_2 (2) no longer have a finite Gröbner basis.

6.3.5 Expected results

Because of knowledge of quadratic modules, we expect the following values, see [9]:

$$\begin{array}{l|l}
A_{n-1} & 1^2 + n^2 \sum_{i=2}^{n-1} \binom{n}{i} \\
B_n \text{ (long root)} & 1^2 + (2n+1)^2 + (2^{2n}) \\
B_n \text{ (short root)} & 1^2 + (2n)^2
\end{array}$$

6.3.6 Lie algebras in other algebras

Sometimes it might be desirable to get a small quotient algebra. C_n has a small quotient algebra if the short root element is set to zero. It is possible to represent an A_{n-1} in this way. For example the the quotient algebra of A_4 has dimension 251, while the quotient algebra of C_5 (shortest root) only has dimension 101.

6.4 Results

Here are some tables of results. The first column is the type of the Lie algebra, the second the dimension of the Lie algebra, the third the root element x for which the relation $x^2 = 0$ was added, the fourth the size of the resulting Gröbner basis and the fifth the dimension of the quotient algebra. If the dimension of the quotient algebra is 1 then only 1 is contained in the quotient algebra (and not the Lie algebra). If the size of the Gröbner basis or quotient algebra is “nf”, then it was found to be infinite. The values for $C_6, 6$ could not be calculated, but are put in the table to indicate that they are different from $C_6, 1$.

The following results are over the rationals:

type	dim	e_i	GB	B	type	dim	e_i	GB	B
A_1	3	1	8	5	C_3	21	1	433	37
A_2	8	1	58	19	C_3	21	3	574	429
A_3	15	1	216	69	C_4	36	1	1284	65
A_4	24	1	613	251	C_4	36	4	3553	4862
A_5	35	1	1550	923	C_5	55	1	3009	101
A_6	48	1	3863	3431	C_5	55	5	29576	58786
B_2	10	1	94	42	C_6	78	1	6064	145
B_2	10	2	97	17	C_6	78	6	?	?
B_3	21	1	461	114	D_4	28	1	834	193
B_3	21	3	436	65	D_5	45	2	2429	613
B_4	36	1	1498	338	D_6	66	1	6220	2193
B_4	36	4	1362	257	E_6	78	1	8002	1459
B_5	55	1	4005	1146	E_7	133	1	24776	1459
B_5	55	5	3612	1025	E_8	248	1	248	1
B_6	78	1	10034	4266	F_4	52	1	52	1
B_6	78	6	9161	4097	F_4	52	2	3443	677
C_2	10	1	96	17	G_2	14	1	14	1
C_2	10	2	94	42	G_2	14	2	196	50

6.4.1 GF(2)

With the added relation $e_i^2 = 0$, the system does not result in a finite set of rules for $A_1, A_2, B_2, B_3, C_3, G_2, D_4, F_4, E_6$.

Using the relation $e_i f_i e_i = e_i$ for all root elements i results in a finite basis for $A_2, A_3, A_4, B_3, B_4, D_4, E_6$ and G_2 . In the case of A_1 the relation $e^2 = 0$ is needed.

type	dim	GB	B
A_1	3	8	5
A_2	8	58	19
A_3	15	216	69
A_4	24	813	251
B_3	21	445	101
B_4	36	1448	321
D_4	28	834	193
E_6	78	1459	834
G_2	14	190	37

6.4.2 GF(3)

Some results over GF(3): Same results for A_n , where $n > 1$, B_n, C_n (small root will give the same result, the large root will lead to an infinite Gröbner basis), D_n, E_6, F_4, G_2 .

type	dim	e_i	GB	B
A_1	3	1	nf	-
A_2	8	1	58	19
B_2	10	1	nf	-
B_2	10	2	97	17
C_2	10	1	97	17
C_2	10	2	nf	-
D_4	28	1	834	193
D_5	45	1	2429	613
E_6	78	1	8002	1459
F_4	52	1	52	1*
F_4	52	4	3443	677
G_2	14	1	28	nf*
G_2	14	2	196	50

6.5 Meataxe

The Meataxe (in GAP see Reference: The MeatAxe) can be used to split up matrix algebras over a finite field. By using a large prime (like 251) it is possible to get information about how it would be split up in the rationals.

```

A1  :1,2,2
A2  :1,3,3,3,3,3,3
B2,1:1,5,5,4,5,5,5,4,4,4
B2,2:1,4,4,4,4
C2,1:1,4,4,4,4
C2,2:1,4,4,5,4,4,5,5,5,5
G2,2:1,7,7,7,7,7,7,7
A3  :1,4,4,6,4,6,4,6,6,6,6,4,4,4,4
B3,3:1,8,8,8,8,8,8,8,8
C3,1:1,6,6,6,6,6,6
C4,1:1,8,8,8,8,8,8,8,8
C5,1:1,10,10,10,10,10,10,10,10,10,10

```

For A_4 in $C_5, 1$ the output is $1, 5 \times 5, 5 \times 5, 5 \times 5, 5 \times 5$ (the same dimension as $C_5, 1$ but it can be split further).

Appendix A

GAP-specific issues

A.1 Introduction

In this chapter several GAP-specific things will be mentioned which are useful when making packages in GAP. They are included as a reference for the maintainers of GBNP.

A.2 Profiling

A.2.1 Introduction

The section of the GAP-manual on profiling says “Profiling can be used to determine in which parts of a program how much time has been spent during runtime”. Thus profiling can be used to learn to how many times certain functions get invocated and how much time is spend there.

This information can be helpful, when one is trying to optimize a GAP program. It is useful to start with the functions where it matters most.

A.2.2 GAP profiling functions

There are several functions mentioned in the GAP-manual, and only the most important will be mentioned here.

- `ProfileFunctions(funcs)` turns profiling on for all functions in the list `funcs`.
- `ProfileGlobalFunctions(true/false)` turns profiling on/off for all global functions (the ones declared with `DeclareGlobalFunction` and installed with `InstallGlobalFunction`).
- `ClearProfile()` clears all stored profiling information.
- `DisplayProfile()` displays all profiling information.
- `DisplayProfile(funcs)` displays the profiling information for functions in the list `funcs`.

A.2.3 Example

The Gröbner basis of the Weyl group of E7 can be calculated in a way comparable with example 3 (Weyl group of E6), with a few extra relations, except the calculation lasts a bit longer. First turn the profiling on:

```
RequirePackage('GBNP');  
  
ProfileFunctions([GBNP.Occur,GBNP.OccurInLst]);  
ProfileGlobalFunctions(true);
```

Now read the file which does the calculations:

```
Read("example103.g");
```

The results can be shown with `DisplayProfile()`:

count	self/ms	chld/ms	function
13944	20	0	Minimum
20512	30	10	Reversed
227555	210	20	GtNP
8370	390	0	List
4184	10	390	LTerms
198960	990	-20	Concatenation
49740	320	1010	Bimul
47964	1980	200	CleanNP
2545180	3160	-150	LtNP
47936	1220	2210	AddNP
2501425	48810	3280	LastReadValue
93352	2900	51890	LastReadValue
1	1710	60050	SGrobner
	61780		TOTAL

Some values are smaller than 0. This is due to inaccuracy. They can be considered to be 0.

Unfortunately the functions from the GBNP record are shown with `LastReadValue` rather than with their real names. To find out which function is meant, one can give the function as an argument, for example `DisplayProfile(GBNP.Occur)`; will output:

count	self/ms	chld/ms	function
2501425	49040	2720	LastReadValue
	12830		OTHER
	61870		TOTAL

So most time goes to `GNP.Occur`.

A.2.4 Profiling record functions

If it is desirable to check all functions in a record then the list of functions can be generated with the following GAP command: `List(RecNames(GBNP), x->GNP.(x))`; . This list can then be given as an argument to `ProfileFunctions`. If desired the profiling information can be displayed separately for each such function:

```
for n in RecNames(GBNP) do
f:=GNP.(n);
if (IsFunction(f)) then
Print(n, "\n"); # We want to know of which function the
# profiling information is.
DisplayProfile(f);
fi;
od;
```

Note that the lookup for record members by strings with the use of brackets is not efficient, but in this case that is not an issue, since this loop will be run only once.

A.3 Methods and objects

Information about methods and objects can be found in the Programmers tutorial, see (Prg Tutorial: Method Selection) and (Prg Tutorial: Creating New Objects). Some advantages of

using objects would be that it is possible to store information (is the object sorted or clean, what is the field, is it a Gröbner basis, is the quotient algebra known, is there a preferred way to print this object).

It is also possible to use functions with the same name and let GAP choose the right one, depending on the attributes of the object, or store the way to print a class of objects.

It might be worthwhile to look into this in the future. Some sheets of an introduction to this can be found at <http://turnbull.mcs.st-and.ac.uk/CIRCA/WkShopTalks/SL/package.pdf>.

Some functions considering the displaying of objects are: `ViewObj` (a short understandable output), `PrintObj` (for reading back in) and `Display` (view in a nicer, formatted format).

A.4 Demo Mode

For presentations it is possible to use the demonstration mode. There is not much documentation about this in the GAP-manual. The first thing to do is to load the file `lib\demo.g` from the GAP directory (use the full path for this). The function `Demonstration` now becomes available which takes as argument a filename to read. By pressing enter the lines from the files will be evaluated one by one.

This has some limitations however. One of them is that errors are not handled very well and that it is impossible to enter more statements on one line. Testing such a demonstration first is recommended.

A.5 Assertions

It is possible to program certain (possible expensive) checks in the code, which only will be evaluated when a certain `AssertionLevel` is high enough. This is similar to the use of `Info` and `InfoLevels`. More information about this can be found at (Reference: Assertions). At the moment no assertions are used in GBNP.

A.6 Compiling

Functions can be compiled to make them a bit faster, see (Reference: The Compiler). For information about which routines may benefit from compiling see (Reference: Suitability for Compilation) (mostly functions which do extensive operations with basic data types, like lists and small integers).

This does not seem to help much in the case of GBNP. It might also be possible to add a few small routines for sublist matching to the kernel of GAP.

A.7 Functions and different argument lengths

It is possible to write a function that can be called with a different number of arguments each time. This can be done by declaring the function with one argument `arg`, which will become a list of the arguments of the function. More about this can be found in (Reference: Function Calls).

A.8 SetHelpViewer

It is possible to view helpfiles in an external program that can view the whole book at once and can follow links, like `xdvi`, `xpdf` or `acroread`. This can be set with the function `SetHelpViewer`. More information about this function can be found at (Reference: SetHelpViewer). This command might be put in the `.gaprc`-file to be executed every time before GAP starts.

A.9 Set additional GAP home directories

Setting GAP-home with `gap -l '~gap/;<System GAP Dir>'` makes it possible to have the source of a package in your own directory (for testing or private usage or if you don't have access to the GAP directory).

Bibliography

- [1] Arjeh M. Cohen and D.A.H. Gijsbers. Noncommutative groebner basis computations. Report, 2003. Available from World Wide Web: <http://www.win.tue.nl/~amc/pub/grobner/gbnp.pdf>. Eindhoven.
- [2] The GAP Group. *GAP – Groups, Algorithms, and Programming, Version 4.3*, 2002. Available from World Wide Web: <http://www.gap-system.org>.
- [3] Edward L. Green. Noncommutative groebner bases and projective resolutions. In *Computational Methods for Representations of Groups and Algebras*, pages 29–60. Birkhäuser, 1999. (Essen 1997).
- [4] Nathan Jacobson. *Lie algebras*. Interscience Tracts in Pure and Applied Mathematics, No. 10. Interscience Publishers (a division of John Wiley & Sons), New York-London, 1962.
- [5] Chris Krook. On the dimension of monomial algebras, 2002.
- [6] S. A. Linton. Constructing matrix representations of finitely presented groups. *J. Symbolic Comput.*, 12(4-5):427–438, 1991. Computational group theory, Part 2.
- [7] S. A. Linton. On vector enumeration. *Linear Algebra Appl.*, 192:235–248, 1993. Computational linear algebra in algebraic and related problems (Essen, 1992).
- [8] Teo Mora. An introduction to commutative and noncommutative Gröbner bases. *Theoretical Computer Science*, 134(1):131–173, 7 November 1994.
- [9] A. A. Premet and I. D. Suprunenko. Quadratic modules for Chevalley groups over fields of odd characteristics. *Math. Nachr.*, 110:65–96, 1983.