ModIsom

A GAP4 Package

Version 2.3.3

by

Bettina Eick

Institut Computational Mathematics, TU Braunschweig

Pockelsstrasse 14, 38106 Braunschweig, Germany

email: beick@tu-bs.de

January 2016

Contents

1 Introduction

This package contains various algorithms related to finite dimensional nilpotent associative algebras. We first give a brief introduction to these algebras and then an overview of the main algorithms.

Associative algebras and nilpotency

Let A be an associative algebra of dimension d over a field F. Let $\{b_1,\ldots,b_d\}$ be a basis for A. We identify the element $x_1b_1+\ldots+x_db_d$ of A with the element $(x_1,\ldots,x_d)$ of F^d. The multiplication of A can then be described by a **structure constants table**: a 3-dimensional array with entries $a_{i,j,k} \in F$ satisfying that

$$b_ib_j = \sum_{k=1}^{d} a_{i,j,k} b_k.$$

An associative algebra A is **nilpotent** if its **power series** terminates at the trivial ideal of A; that is

$$A > A^2 > \ldots > A^c > A^{c+1} = \{0\}$$

where A^j is the ideal of A generated by all products of length at least j. The length c of the power series is also called the **class** of A and the dimension of A/A^2 is the **rank** of A. Note that A is generated by $dim(A/A^2)$ elements. Clearly, A does not contain a multiplicative identity.

For computational purposes we describe a nilpotent associative algebra by a weighted basis and a description of the corresponding structure constants table. A basis of a nilpotent associative algebra A is **weighted** if there is a sequence of weights $(w_1,\ldots,w_d)$ so that

$$A^j = \langle b_i \mid w_i \geq j \rangle.$$

Note that $AA^j = A^{j+1}$ for every j. Thus it is possible to choose all basis elements of weight at least 2 so that $b_i = b_kb_l$ holds for some k and l, where b_k is of weight 1 and b_l is of weight $w_i - 1$. This feature allows an effective description of A via a **nilpotent structure constants table**. This contains the structure constants $a_{i,j,k}$ for all i with $w_i = 1$ and $1 \leq j,k \leq d$. For i with $w_i > 1$ it either contains a description as $b_i = b_kb_l$ or the structure constants $a_{i,j,k}$ for $1 \leq j,k \leq d$. It may also contain both or some partial overlap of these informations.

Isomorphisms and Automorphisms

Let A be a finite dimensional nilpotent associative algebra over a finite field. This package contains an implementation of the methods in [Eic08] which allow the determination of the automorphism group $Aut(A)$ and a **canonical form** $Can(A)$.

The automorphism group is given by generators and it represented as a subgroup of $GL(dim(A),F)$. Also the order of $Aut(A)$ is available.

A canonical form $Can(A)$ for A is a nilpotent structure constants table for A which is unique for the isomorphism type of A; that is, two algebras A and B are isomorphic if and only if $Can(A) = Can(B)$ holds. Hence the canonical form can be used to solve the isomorphism problem.

The modular isomorphism problem

The modular isomorphism problem asks whether $\mathbb{F}G \cong \mathbb{F}H$ implies that $G \cong H$ for two p-groups G and H and $\mathbb{F}$ the field with p elements. This problem is still open, despite various efforts towards proving the claim or finding counterexamples to it.

Computational approaches have been used to investigate the modular isomorphism problem. Based on an algorithm by Roggenkamp and Scott [RS93], Wursthorn [Wur93] described an algorithm for checking the modular isomorphism problem; that is, he described an algorithm for checking whether two modular group algebras $\mathbb{F}G$ and $\mathbb{F}H$ are isomorphic. This algorithm has been implemented in C by Wursthorn and has been used applied to the groups of order dividing 2^7 without finding a counterexample, see [BKRW99].

This package contains an implementation of the new algorithm described in [Eic08] for checking isomorphism of modular group algebras. It is based on the fact that the Jacobson radical $J(FG)$ is nilpotent if FG is a modular group algebra. Hence the automorphism group and canonical form algorithm of this package apply and can be used to solve the isomorphism problem for modular group algebras.

The methods of this package have been used to check the modular isomorphism problem for the groups of order dividing 3^6 and 2^8 ([Eic08]) and for the groups of order 2^9 ([EKo11]).

A nilpotent quotient algorithm

Given a finitely presented associative algebra A over an arbitrary field F, this package contains an algorithm to determine a nilpotent structure constants table for the class-c nilpotent quotient of A. See [Eic11] for details on the underlying algorithm.

Kurosh Algebras

Let $F(d,F)$ denote the free non-unital associative algebra on d generators over the field F. Then

$$A(d,n,F) = F(d,F)/\langle\langle w^n \mid w \in F(d,F)\rangle\rangle$$

is the **Kurosh Algebra** on d generators of exponent n over the field F. Kurosh Algebras can be considered as an algebra-theoretic analogue to Burnside groups.

This package contains a method that allows to determine $A(d,n,F)$ for given d,n,F. This can also be used to determine $A(d,n,F)$ for all fields of a given characteristic. We refer to [Eic11] for details on the algorithms.

This package also contains a database of Kurosh Algebras that have been determined with the methods of this package.

2 Tables

Finite dimensional algebras can be described by structure contants tables. For nilpotent algebras it is not neccessary to store a full structure contants table. To use this feature, we introduce **nilpotent structure constants tables** or just **nilpotent tables** for short. These are used heavily throughout the package.

2.1 Nilpotent tables

Let A be a finite-dimensional nilpotent associative algebra over a field F. Let $(b_1, \ldots, b_d)$ be a **weighted basis** of A; that is, a basis with weights $(w_1, \ldots, w_d)$ satifying that $A^j = \langle b_i \mid w_i \geq j \rangle$. Let

$$b_i b_j = \sum_k a_{i,j,k} b_k.$$

The nilpotent table T for A (with respect to the basis $(b_1, \ldots, b_d)$) is a record with the following entries.

dim
the dimension d of A;

fld
the field F of A;

wgs
the weights $(w_1, \ldots, w_d)$;

rnk
the rank e of A (i.e. the dimension of A/A^2).

wds
a list of length d with holes; If the ith entry is bounded, then it is of the form $[k, l]$. In this case, $w_i > 1$ and $b_i = b_k b_l$ and $w_k = 1$ and $w_l = w_i - 1$ holds.

tab
a partial structure contants table for A; If $tab[i][j][k]$ is bounded, then it is $a_{i,j,k}$. Note that either a full vector $tab[i][j]$ is given or $tab[i][j]$ is unbounded. The entry $tab[i][j][k]$ is available for $1 \leq i,j \leq e$ and if $wds[i]$ is unbounded.

com
optional; If this is bounded, then it is a boolean. If this boolean is true, then the algebra is assumed to be commutative.

In a nilpotent table not all structure contants are readily available. The following function determines the structure constants for the product $b_i b_j$. If the global variable *STORE* is true, then the function stores the computed entry in the table.

1 ▶ `GetEntryTable( T, i, j )` F

We consider two nilpotent tables as equal, if they would be equal if the full set of structure constants tables would be bound. The following function provides an effective check for this.

2 ► `CompareTables( T1, T2 )` F

A nilpotent table contains redundant information and hence can be inconsistent. The next functions can be used to check this to some extend.

3 ► `CheckAssociativity( T )` F

Checks that $(b_ib_j)b_k = b_i(b_jb_k)$ for all i, j, k. Note that this may be time-consuming.

4 ► `CheckCommutativity( T )` F

Checks whether *T* defines a commutative algebra and sets the entry *com* accordingly.

5 ► `CheckConsistency( T )` F

Checks that *wds* and *tab* are compatible. This assumes that CheckAssociativity returns true.

All later described algorithms of this package assume that the tables considered are fully consistent.

```
  gap> T := rec( dim := 3,
                 fld := GF(2),
                 rnk := 2,
                 wgs := [ 1, 1, 2 ],
                 wds := [ ,, [ 2, 1 ] ],
                 tab := [] );;
  gap> T.tab[1] := [[0,0,0],[0,0,1]] * One(T.fld);;
  gap> T.tab[2] := [[0,0,1],[0,0,0]] * One(T.fld);;
  gap> GetEntryTable( T, 3, 1 );
[ 0*Z(2), 0*Z(2), 0*Z(2) ]
```

2.2 Algebras in the GAP sense

We provide functions to convert back and forth between algebras in the GAP sense and nilpotent tables.

1 ► `AlgebraByTable( T )` F
► `NilpotentTable( A )` F

Note that the second function fails if *A* is not nilpotent.

For modular group algebras of p-groups, the group algebra itself is not nilpotent (as it contains a unit), but its Jacobson radial is. The following function determines a nilpotent table for the Jacobson radical.

2 ► `NilpotentTableOfRad( FG )` F

```
  gap> A := GroupRing(GF(2), SmallGroup(8,3));
  <algebra-with-one over GF(2), with 3 generators>
  gap> NilpotentTableOfRad(A);
  rec( dim := 7, fld := GF(2), rnk := 2,
    tab :=
      [
        [ [ 0*Z(2), 0*Z(2), 0*Z(2), 0*Z(2), 0*Z(2), 0*Z(2), 0*Z(2) ],
            [ 0*Z(2), 0*Z(2), Z(2)^0, 0*Z(2), 0*Z(2), 0*Z(2), 0*Z(2) ],
            [ 0*Z(2), 0*Z(2), 0*Z(2), 0*Z(2), 0*Z(2), 0*Z(2), 0*Z(2) ],
            [ 0*Z(2), 0*Z(2), 0*Z(2), 0*Z(2), Z(2)^0, 0*Z(2), 0*Z(2) ],
            [ 0*Z(2), 0*Z(2), 0*Z(2), 0*Z(2), 0*Z(2), 0*Z(2), 0*Z(2) ],
            [ 0*Z(2), 0*Z(2), 0*Z(2), 0*Z(2), 0*Z(2), 0*Z(2), Z(2)^0 ],
            [ 0*Z(2), 0*Z(2), 0*Z(2), 0*Z(2), 0*Z(2), 0*Z(2), 0*Z(2) ] ],
        [ [ 0*Z(2), 0*Z(2), Z(2)^0, Z(2)^0, Z(2)^0, Z(2)^0, Z(2)^0 ],
            [ 0*Z(2), 0*Z(2), 0*Z(2), 0*Z(2), 0*Z(2), 0*Z(2), 0*Z(2) ],
            [ 0*Z(2), 0*Z(2), 0*Z(2), 0*Z(2), 0*Z(2), Z(2)^0, Z(2)^0 ],
            [ 0*Z(2), 0*Z(2), 0*Z(2), 0*Z(2), 0*Z(2), Z(2)^0, 0*Z(2) ],
            [ 0*Z(2), 0*Z(2), 0*Z(2), 0*Z(2), 0*Z(2), 0*Z(2), Z(2)^0 ],
            [ 0*Z(2), 0*Z(2), 0*Z(2), 0*Z(2), 0*Z(2), 0*Z(2), 0*Z(2) ],
            [ 0*Z(2), 0*Z(2), 0*Z(2), 0*Z(2), 0*Z(2), 0*Z(2), 0*Z(2) ] ],,
        [ [ 0*Z(2), 0*Z(2), 0*Z(2), 0*Z(2), Z(2)^0, 0*Z(2), 0*Z(2) ],
            [ 0*Z(2), 0*Z(2), 0*Z(2), 0*Z(2), 0*Z(2), Z(2)^0, 0*Z(2) ],
            [ 0*Z(2), 0*Z(2), 0*Z(2), 0*Z(2), 0*Z(2), 0*Z(2), Z(2)^0 ],
            [ 0*Z(2), 0*Z(2), 0*Z(2), 0*Z(2), 0*Z(2), 0*Z(2), 0*Z(2) ],
            [ 0*Z(2), 0*Z(2), 0*Z(2), 0*Z(2), 0*Z(2), 0*Z(2), 0*Z(2) ],
            [ 0*Z(2), 0*Z(2), 0*Z(2), 0*Z(2), 0*Z(2), 0*Z(2), 0*Z(2) ],
            [ 0*Z(2), 0*Z(2), 0*Z(2), 0*Z(2), 0*Z(2), 0*Z(2), 0*Z(2) ] ] ],
    wds := [ ,, [ 1, 2 ],, [ 1, 4 ], [ 2, 4 ], [ 1, 6 ] ],
 wgs := [ 1, 1, 2, 2, 3, 3, 4 ] )
```

3 Automorphism groups and Canonical Forms

We refer to [Eic08] for background on the algorithms used in this Chapter. Throughout the chapter, we assume that F is a finite field.

3.1 Automorphism groups

Let T be a nilpotent table over F. The following function can be used to determine the automorphism group of the algebra described by T. The automorphism group is determined as subgroup of $GL(T \cdot dim, T \cdot fld)$ given by generators and its order. There is a variation available to determine the automorphism group of a modular group algebra FG, where F is a finite field and G is a p-group.

1 ▶ `AutGroupOfTable( T )` F
▶ `AutGroupOfRad( FG )` F

In both cases, the automorphism group is described by a record. The matrices in the lists *glAutos* and *agAutos* generate together the automorphism group. The matrices in *agAutos* generate a p-group. The entry *size* contains the order of the automorphism group.

3.2 Canonical forms

Let T be a nilpotent table. The following function can be used to determine the automorphism group of T if the underlying field of T is finite. The canonical form is a nilpotent table which is unique for the isomorphism type of the algebra defined by T. Again there a variation available for modular group algebras.

1 ▶ `CanonicalFormOfTable( T )` F
▶ `CanonicalFormOfRad( FG )` F

The automorphism group of T is determined as a side-product of computing the canonical form. The following functions can be used to return both.

2 ▶ `CanoFormWithAutGroupOfTable( T )` F
▶ `CanoFormWithAutGroupOfRad( FG )` F

In both cases, these functions return a record with entries *cano* and *auto*.

3.3 Example of canonical form computation

We compute the automorphism group and a canonical form for the modular group algebra of the dihedral group of order 8.

```
gap> A := GroupRing(GF(2), SmallGroup(8,3));;
gap> T := TableByWeightedBasisOfRad(A);;
gap> C := CanoFormWithAutGroupOfTable(T);;

# check that the canonical form is not equal to T
gap> CompareTables(C.cano, T);
false
```

```
# the order of the automorphism group
gap> C.auto.size;
512

# the entries of the canonical table as far as they are bounded
gap> C.cano.tab;
[ [ <a GF2 vector of length 7>, <a GF2 vector of length 7>,
      [ 0*Z(2), 0*Z(2), 0*Z(2), 0*Z(2), Z(2)^0, 0*Z(2), 0*Z(2) ],
      [ 0*Z(2), 0*Z(2), 0*Z(2), 0*Z(2), 0*Z(2), 0*Z(2), 0*Z(2) ],
      [ 0*Z(2), 0*Z(2), 0*Z(2), 0*Z(2), 0*Z(2), 0*Z(2), 0*Z(2) ],
      [ 0*Z(2), 0*Z(2), 0*Z(2), 0*Z(2), 0*Z(2), 0*Z(2), Z(2)^0 ],
      [ 0*Z(2), 0*Z(2), 0*Z(2), 0*Z(2), 0*Z(2), 0*Z(2), 0*Z(2) ] ],
  [ <a GF2 vector of length 7>, <a GF2 vector of length 7>,
      [ 0*Z(2), 0*Z(2), 0*Z(2), 0*Z(2), 0*Z(2), 0*Z(2), 0*Z(2) ],
      [ 0*Z(2), 0*Z(2), 0*Z(2), 0*Z(2), 0*Z(2), Z(2)^0, 0*Z(2) ],
      [ 0*Z(2), 0*Z(2), 0*Z(2), 0*Z(2), 0*Z(2), 0*Z(2), Z(2)^0 ],
      [ 0*Z(2), 0*Z(2), 0*Z(2), 0*Z(2), 0*Z(2), 0*Z(2), 0*Z(2) ],
      [ 0*Z(2), 0*Z(2), 0*Z(2), 0*Z(2), 0*Z(2), 0*Z(2), 0*Z(2) ] ],
  [ [ 0*Z(2), 0*Z(2), 0*Z(2), 0*Z(2), 0*Z(2), 0*Z(2), 0*Z(2) ],
      [ 0*Z(2), 0*Z(2), 0*Z(2), 0*Z(2), 0*Z(2), Z(2)^0, 0*Z(2) ],
      [ 0*Z(2), 0*Z(2), 0*Z(2), 0*Z(2), 0*Z(2), 0*Z(2), Z(2)^0 ],
      [ 0*Z(2), 0*Z(2), 0*Z(2), 0*Z(2), 0*Z(2), 0*Z(2), 0*Z(2) ] ],
  [ [ 0*Z(2), 0*Z(2), 0*Z(2), 0*Z(2), Z(2)^0, 0*Z(2), 0*Z(2) ],
      [ 0*Z(2), 0*Z(2), 0*Z(2), 0*Z(2), 0*Z(2), 0*Z(2), 0*Z(2) ],
      [ 0*Z(2), 0*Z(2), 0*Z(2), 0*Z(2), 0*Z(2), 0*Z(2), 0*Z(2) ],
      [ 0*Z(2), 0*Z(2), 0*Z(2), 0*Z(2), 0*Z(2), 0*Z(2), Z(2)^0 ] ],
  [ [ 0*Z(2), 0*Z(2), 0*Z(2), 0*Z(2), 0*Z(2), 0*Z(2), 0*Z(2) ],
      [ 0*Z(2), 0*Z(2), 0*Z(2), 0*Z(2), 0*Z(2), 0*Z(2), Z(2)^0 ] ],
  [ [ 0*Z(2), 0*Z(2), 0*Z(2), 0*Z(2), 0*Z(2), 0*Z(2), Z(2)^0 ],
   [ 0*Z(2), 0*Z(2), 0*Z(2), 0*Z(2), 0*Z(2), 0*Z(2), 0*Z(2) ] ] ]
```

4 The modular isomorphism problem

An application of the methods in this package has been the checking of the modular isomorphism problems for the groups of order dividing 2^8, 3^6 and 2^9 [Eic07,EKo10]. This section contains the functions used for this purpose.

4.1 Computing and checking bins

1 ► `BinsByGT( p, n )` F

returns a partion of the list $[1 \cdot \cdot \mathit{NumberSmallGroups}(p^n)]$ into sublists so that the modular group algebras of two groups SmallGroup(p^n, i) and SmallGroup(p^n, j) can not be isomorphic if i and j are in different lists. The function BinsByGT uses various group theoretic invariants to split the groups of order p^n in bins.

2 ► `CheckBin( p, n, k, bin )` F

For $i \in \mathit{bin}$ let G_i denote SmallGroup(p^n, i) and let A_i be the augementation ideal of FG_i. This function computes and compares the canonical forms of the algebras A_i/A_i^j for every $i \in \mathit{bin}$ and increasing $j \in \{1, \dots, k+1\}$.

At each level j it splits the current bins into sub-bins according to the different canonical forms of A_i/A_i^j. Bins of length 1 are then discarded.

The function returns if no further bins are available or if $j = k+1$ is reached. In the later case the function returns the remaining bins.

4.2 Example for groups of order 64

We show how to check the modular isomorphism problem for the groups of order 64. We first use BinsByGT to determine bins and we then check the first of the resulting bins with CheckBin. The fact that CheckBin ends with an empty list of bins shows that all groups are splitted.

```
gap> bins := BinsByGT(2,6);
refine by abelian invariants of group (Sehgal/Ward)
13 bins with 256 groups
refine by abelian invariants of center (Sehgal/Ward)
30 bins with 237 groups
refine by lower central series (Sandling)
32 bins with 127 groups
refine by jennings series (Passi+Sehgal/Ritter+Sehgal)
36 bins with 123 groups
refine by conjugacy classes (Roggenkamp/Wursthorn)
16 bins with 36 groups
refine by elem-ab subgroups (Quillen)
  start bin 1 of 16
  start bin 2 of 16
  start bin 3 of 16
  start bin 4 of 16
  start bin 5 of 16
```

```
    start bin 6 of 16
    start bin 7 of 16
    start bin 8 of 16
    start bin 9 of 16
    start bin 10 of 16
    start bin 11 of 16
    start bin 12 of 16
    start bin 13 of 16
    start bin 14 of 16
    start bin 15 of 16
    start bin 16 of 16
  9 bins with 21 groups
  [ [ 13, 14 ], [ 18, 19 ], [ 20, 22 ], [ 97, 101 ], [ 108, 110 ],
    [ 155, 157, 159 ], [ 156, 158, 160 ], [ 173, 176 ], [ 179, 180, 181 ] ]

  gap> CheckBin(2,6,100,bins[1]);
  compute tables through power series
    determined table for 1
    determined table for 2

  refine bin
    weights yields bins [ [ 1, 2 ] ]
    layer 1 yields bins [ [ 1, 2 ] ]
    layer 2 yields bins [ [ 1, 2 ] ]
    layer 3 yields bins [ [ 1, 2 ] ]
 layer 4 yields bins [  ]
```

5 Nilpotent Quotients

This chapter contains a description of the nilpotent quotient algorithm for associative finitely presented algebras. We refer to [Eic11] for background on the algorithms used in this Chapter.

5.1 Computing nilpotent quotients

Let A be a finitely presented algebra in the GAP sense. The following function can be used to determine the class-c nilpotent quotient of A. The quotient is described by a nilpotent table.

1 ▶ `NilpotentQuotientOfFpAlgebra( A, c )` F

The output of this function is a nilpotent table with some additional entries. In particular, there is the additional entry *img* which describes the images of the generators of A in the nilpotent table.

5.2 Example of nilpotent quotient computation

```
gap> F := FreeAssociativeAlgebra(GF(2), 2);;
gap> g := GeneratorsOfAlgebra(F);;
gap> r := [g[1]^2, g[2]^2];;
gap> A := F/r;;
gap> NilpotentQuotientOfFpAlgebra(A,3);
rec( def := [ 1, 2 ], dim := 8, fld := GF(2),
  img := [ <a GF2 vector of length 8>, <a GF2 vector of length 8> ],
  mat := [ [  ], [  ] ], rnk := 2,
  tab :=
    [ [<a GF2 vector of length 8>, <a GF2 vector of length 8>,
        [ 0*Z(2), 0*Z(2), 0*Z(2), 0*Z(2), Z(2)^0, 0*Z(2), 0*Z(2), 0*Z(2) ],
        [ 0*Z(2), 0*Z(2), 0*Z(2), 0*Z(2), 0*Z(2), 0*Z(2), 0*Z(2), 0*Z(2) ],
        [ 0*Z(2), 0*Z(2), 0*Z(2), 0*Z(2), 0*Z(2), 0*Z(2), 0*Z(2), 0*Z(2) ],
        [ 0*Z(2), 0*Z(2), 0*Z(2), 0*Z(2), 0*Z(2), 0*Z(2), 0*Z(2), Z(2)^0 ],
        [ 0*Z(2), 0*Z(2), 0*Z(2), 0*Z(2), 0*Z(2), 0*Z(2), 0*Z(2), 0*Z(2) ],
        [ 0*Z(2), 0*Z(2), 0*Z(2), 0*Z(2), 0*Z(2), 0*Z(2), 0*Z(2), 0*Z(2) ] ],
      [ <a GF2 vector of length 8>, <a GF2 vector of length 8>,
        [ 0*Z(2), 0*Z(2), 0*Z(2), 0*Z(2), 0*Z(2), 0*Z(2), 0*Z(2), 0*Z(2) ],
        [ 0*Z(2), 0*Z(2), 0*Z(2), 0*Z(2), 0*Z(2), Z(2)^0, 0*Z(2), 0*Z(2) ],
        [ 0*Z(2), 0*Z(2), 0*Z(2), 0*Z(2), 0*Z(2), 0*Z(2), Z(2)^0, 0*Z(2) ],
        [ 0*Z(2), 0*Z(2), 0*Z(2), 0*Z(2), 0*Z(2), 0*Z(2), 0*Z(2), 0*Z(2) ],
        [ 0*Z(2), 0*Z(2), 0*Z(2), 0*Z(2), 0*Z(2), 0*Z(2), 0*Z(2), 0*Z(2) ],
        [ 0*Z(2), 0*Z(2), 0*Z(2), 0*Z(2), 0*Z(2), 0*Z(2), 0*Z(2), 0*Z(2) ] ],
      [ [ 0*Z(2), 0*Z(2), 0*Z(2), 0*Z(2), 0*Z(2), 0*Z(2), 0*Z(2), 0*Z(2) ],
        [ 0*Z(2), 0*Z(2), 0*Z(2), 0*Z(2), 0*Z(2), Z(2)^0, 0*Z(2), 0*Z(2) ] ],
      [ [ 0*Z(2), 0*Z(2), 0*Z(2), 0*Z(2), Z(2)^0, 0*Z(2), 0*Z(2), 0*Z(2) ],
        [ 0*Z(2), 0*Z(2), 0*Z(2), 0*Z(2), 0*Z(2), 0*Z(2), 0*Z(2), 0*Z(2) ] ]],
  wds := [ ,, [ 2, 1 ], [ 1, 2 ], [ 1, 3 ], [ 2, 4 ], [ 2, 5 ], [ 1, 6 ] ],
wgs := [ 1, 1, 2, 2, 3, 3, 4, 4 ] )
```

6 Relatively free Algebras

As described in [Eic11], the nilpotent quotient algorithm also allows to determine certain relatively free algebras; that is, algebras that are free within a variety.

6.1 Computing Kurosh Algebras

1 ► `KuroshAlgebra( d, n, F )` F

determines a nilpotent table for the largest associative algebra on d generators over the field F so that every element a of the algebra satisfies $a^n = 0$.

2 ► `ExpandExponentLaw( T, n )`

suppose that T is the nilpotent table of a Kurosh algebra of exponent n defined over a prime field. This function determines polynomials describing the corresponding Kurosh algebras over all fields with the same characteristic as the prime field.

6.2 A Library of Kurosh Algebras

The package contains a library of Kurosh algebras. This can be accessed as follows.

1 ► `KuroshAlgebraByLib(d, n, F)` F

At current, the library contains the Kurosh algebras for $n = 2$, $(d,n) = (2,3)$, $(d,n) = (3,3)$ and $F = \mathbb{Q}$ or $|F| \in \{2,3,4\}$, $(d,n) = (4,3)$ and $F = \mathbb{Q}$ or $|F| \in \{2,3,4\}$, $(d,n) = (2,4)$ and $F = \mathbb{Q}$ or $|F| \in \{2,3,4,9\}$, $(d,n) = (2,5)$ and $F = \mathbb{Q}$ or $|F| \in \{2,3,4,5,8,9\}$.

6.3 Example of accessing the library of Kurosh algebras

```
 gap> KuroshAlgebra(2,2,Rationals);
 ... some printout ..
 rec( bas := [ [ 1, 0, 0, 0 ], [ 0, 1, 1, 0 ], [ 0, 0, 0, 1 ], [ 0, 1, 0, 0 ] ]
     , com := false, dim := 3, fld := Rationals, rnk := 2,
   tab := [ [ [ 0, 0, 0 ], [ 0, 0, -1 ], [ 0, 0, 0 ] ],
       [ [ 0, 0, 1 ], [ 0, 0, 0 ], [ 0, 0, 0 ] ] ], wds := [ ,, [ 2, 1 ] ],
wgs := [ 1, 1, 2 ] )
```

Bibliography

[BKRW99] Frauke M. Bleher, Wolfgang Kimmerle, Klaus W. Roggenkamp, and Martin Wursthorn. Computational aspects of the isomorphism problem. In *Algorithmic algebra and number theory (Heidelberg, 1997)*, pages 313–329. Springer, Berlin, 1999.

[Eic08] Bettina Eick. Computing automorphism groups and testing isomorphisms for modular group algebras. *J. Algebra*, 320(11):3895–3910, 2008.

[Eic11] Bettina Eick. A nilpotent quotient algorithm for finitely presented associative algebras and algebras satisfying a polynomial identity. *Accepted for IJAC*, 2011.

[EK11] Bettina Eick and Alexander Konovalov. The modular isomorphism problem for the groups of order 2^9. *In Proceedings of 'Groups St. Andrews' 2009*, 2011.

[RS93] K. W. Roggenkamp and L. L. Scott. Automorphisms and nonabelian cohomology: an algorithm. *Linear Algebra Appl.*, 192:355–382, 1993.

[Wur93] Martin Wursthorn. Isomorphisms of modular group algebras: an algorithm and its application to groups of order 2^6. *J. Symbolic Comput.*, 15(2):211–227, 1993.