Handbook of Magma Functions

Volume 7

Algebras

John Cannon Wieb Bosma
Claus Fieker Allan Steel

Editors

Version 2.19
Sydney
December 17, 2012

MAGMA
COMPUTER ● ALGEBRA

Handbook of Magma Functions

Editors:

John Cannon   Wieb Bosma   Claus Fieker   Allan Steel

Handbook Contributors:

Geoff Bailey, Wieb Bosma, Gavin Brown, Nils Bruin, John Cannon, Jon Carlson, Scott Contini, Bruce Cox, Brendan Creutz, Steve Donnelly, Tim Dokchitser, Willem de Graaf, Andreas-Stephan Elsenhans, Claus Fieker, Damien Fisher, Volker Gebhardt, Sergei Haller, Michael Harrison, Florian Hess, Derek Holt, David Howden, Al Kasprzyk, Markus Kirschmer, David Kohel, Axel Kohnert, Dimitri Leemans, Paulette Lieby, Graham Matthews, Scott Murray, Eamonn O'Brien, Dan Roozemond, Ben Smith, Bernd Souvignier, William Stein, Allan Steel, Damien Stehlé, Nicole Sutherland, Don Taylor, Bill Unger, Alexa van der Waall, Paul van Wamelen, Helena Verrill, John Voight, Mark Watkins, Greg White

Production Editors:

Wieb Bosma   Claus Fieker   Allan Steel   Nicole Sutherland

HTML Production:

Claus Fieker   Allan Steel

VOLUME 7: OVERVIEW

VOLUME 7: CONTENTS

XII REPRESENTATION THEORY 2681

PART XI
ALGEBRAS

79 ALGEBRAS

Chapter 79
ALGEBRAS

79.1 Introduction

Algebras are viewed as free modules over a ring R with an additional multiplication. There are no a priori conditions imposed on the ring except that it must be unital, but some functions may require that an echelonization algorithm is available for modules over R and sometimes it is also required that R is a field. For example, quotients of algebras can only be constructed over fields, since otherwise the quotient module is not necessarily a free module over R.

The most general way to define an algebra is by structure constants, but for special types of algebras MAGMA uses more efficient representations.

79.1.1 The Categories of Algebras

At present, MAGMA contains seven main categories of algebras:

(1) General algebras represented by structure constants: category `AlgGen`;

(2) Associative algebras represented by structure constants: category `AlgAss`;

(3) Quaternion algebras as special types of associative algebras; category `AlgQuat`;

(4) Lie algebras represented by structure constants: category `AlgLie`;

(5) Group algebras: category `AlgGrp` with a special type `AlgGrpSub` for subalgebras of group algebras;

(6) Matrix algebras: category `AlgMat`;

(7) Finitely presented algebras: category `AlgFP`.

The hierarchy of these categories is such that `AlgGen` is on the top level and `AlgAss` and `AlgLie` are on the next level inheriting the functions available for `AlgGen`. The categories `AlgQuat`, `AlgGrp` and `AlgMat` are on a third level inheriting the functions available for `AlgAss`. Finitely presented algebras are independent of the other categories.

79.2 Construction of General Algebras and their Elements

79.2.1 Construction of a General Algebra

The construction of an algebra depends on its category. The chapters on the individual algebra categories describe this in detail. Here only an overview is given.

```
Algebra< R, n | Q >
```

Let R be ring, n an integer and Q a sequence of n^3 elements of R. This function creates an algebra A of dimension n over R with basis $e_1, \ldots, e_n$ such that Q contains the structure constants of A, i.e. $e_i * e_j = \sum a_{ij}^k e_k$, where a_{ij}^k is the element in position $(i-1)*n^2+(j-1)*n+k$ of Q.

```
AssociativeAlgebra< R, n | Q >
```

`Check` BoolElt *Default :* `true`

This function creates the associative structure constant algebra A as returned by `Algebra< R, n | Q >`. By default, the algebra is checked on associativity, but this can be avoided by setting `Check := false`. The returned algebra is of type `AlgAss`.

```
QuaternionAlgebra< K | a, b >
```

This function creates the quaternion algebra A over the field K on generators x and y with relations $x^2 = a$, $y^2 = b$, and $xy = -yx$.

```
LieAlgebra< R, n | Q >
```

`Check` BoolElt *Default :* `true`

This function creates the Lie structure constant algebra A as returned by `Algebra< R, n | Q >`. By default, the algebra is checked to be a Lie algebra, but this can be avoided by setting `Check := false`. The returned algebra is of type `AlgLie`.

```
LieAlgebra(A)
```

Given an associative algebra A, create the Lie algebra generated by the elements in L using the induced Lie product $(x, y) \to x*y - y*x$.

```
GroupAlgebra(R, G)
```

Given a ring R and a group G construct the group algebra $R[G]$ of dimension $|G|$ over R.

```
MatrixAlgebra(R, n)
```

Given a positive integer n and a ring R, create the full matrix algebra $M_n(R)$ of dimension n^2 over R.

79.2.2 Construction of an Element of a General Algebra

The construction of a generic element of an algebra varies for the different types of algebras and is therefore explained in the corresponding chapters.

```
Zero(A)
```

```
A ! 0
```

Create the zero element of the algebra A.

```
One(A)
```

```
A ! 1
```

If it exists, create the identity element of the algebra A; otherwise an error occurs.

```
Random(A)
```

Given an algebra A defined over a finite ring, return a random element.

79.3 Construction of Subalgebras, Ideals and Quotient Algebras

79.3.1 Subalgebras and Ideals

If the coefficient ring R of an algebra A is a Euclidean domain then one may construct submodules and ideals of A in Magma.

```
sub< A | L >
```

Create the subalgebra S of the algebra A that is generated by the elements defined by L, where L is a list of one or more items of the following types:

(a) An element of A;

(b) A set or sequence of elements of A;

(c) A subalgebra or ideal of A;

(d) A set or sequence of subalgebras or ideals of A.

The constructor returns the subalgebra as an algebra of the same type as A. An exception are group algebras, where the subalgebra is either of type `AlgAss` or of the special type `AlgGrpSub`. As well as the subalgebra S itself, the constructor returns the inclusion homomorphism $f : S \to A$.

```
lideal< A | L >
```

Create the left ideal I of the algebra A generated by the elements defined by L, where L is a list as for the `sub` constructor above.

The constructor returns the left ideal as an algebra of the same type as A with the same exception for group algebras as for the `sub` constructor. As well as the left ideal I itself, the constructor returns the inclusion homomorphism $f : I \to A$.

`rideal< A | L >`

Create the right ideal I of the algebra A generated by the elements defined by L, where L is a list as for the `sub` constructor above.

The constructor returns the right ideal as an algebra of the same type as A with the same exception for group algebras as for the `sub` constructor. As well as the right ideal I itself, the constructor returns the inclusion homomorphism $f : I \to A$.

`ideal< A | L >`

Create the (two-sided) ideal I of the algebra A generated by the elements defined by L, where L is a list as for the `sub` constructor above.

The constructor returns the right ideal as an algebra of the same type as A with the same exception for group algebras as for the `sub` constructor. As well as the ideal I itself, the constructor returns the inclusion homomorphism $f : I \to A$.

79.3.2 Quotient Algebras

If the coefficient ring R of an algebra A is a field, then quotient algebras of A may also be constructed.

`quo< A | L >`

Create the quotient algebra $Q = A/I$, where I is the two-sided ideal of A generated by the elements of A specified by the list L, which should satisfy the same conditions as for the `sub` constructor above.

The constructor returns the quotient as a structure constant algebra with degree equal to its dimension. If A is known to be associative, then Q is of type `AlgAss`, otherwise Q is of type `AlgGen`. As well as the quotient Q itself, the constructor returns the natural homomorphism $f : A \to Q$.

`A / S`

The quotient of the algebra A by the (two-sided) ideal closure of its subalgebra S.

79.4 Operations on Algebras and Subalgebras

79.4.1 Invariants of an Algebra

`CoefficientRing(A)`

`BaseRing(A)`

The coefficient ring (or base ring) over which the algebra A is defined.

`Dimension(A)`

The dimension of the algebra A.

`#A`

The cardinality of the algebra A if both R and the dimension of A are finite. Note that this cannot be computed if the dimension of A is too large.

79.4.2 Changing Rings

```
ChangeRing(A, S)
```

Given an algebra A with base ring R, together with a ring S, construct the algebra B with base ring S obtained by coercing the coefficients of elements of A into S, together with the homomorphism from A to B.

This function can not be applied if A is of type `AlgGrpSub`, as the parent structure of elements of A is the full group algebra of which A is a subalgebra.

```
ChangeRing(A, S, f)
```

Given an algebra A with base ring R, together with a ring S and a map $f : R \rightarrow S$, construct the algebra B with base ring S obtained by mapping the coefficients of elements of A into S via f, together with the homomorphism from A to B.

As above, this function can not be applied if A is of type `AlgGrpSub`.

79.4.3 Bases

In general, every algebra comes with a basis, corresponding to its underlying module structure. The only exception for that are group algebras in the `"Terms"` representation, where the dimension of the algebra may be too large to create vectors of that degree.

```
BasisElement(A, i)
A . i
```

The i-th basis element of the algebra A.

```
Basis(A)
```

The basis of the algebra A, as a sequence of elements of A.

Note that if A is of type `AlgGrpSub` the returned elements will be elements of the full group algebra of which A is a subalgebra.

```
IsIndependent(Q)
```

Given a sequence Q of elements of the R-algebra A, this functions returns `true` if these elements are linearly independent over R; otherwise `false`.

```
ExtendBasis(S, A)
ExtendBasis(Q, A)
```

Given an algebra A and either a subalgebra S of dimension m of A or a sequence Q of m linearly independent elements of A, return a sequence containing a basis of A such that the first m elements are the basis of S resp. the elements in Q.

79.4.4 Decomposition of an Algebra

An algebra A can be regarded as a (left- or right-) module for itself. If A is defined over a finite field, the machinery to decompose modules over finite fields can be used to investigate the structure of the algebra A.

`CompositionSeries(A)`

Compute a composition series for the algebra A. The function has three return values:

(a) a sequence containing the composition series as an ascending chain of subalgebras such that the successive quotients are irreducible A-modules;

(b) a sequence containing the composition factors as structure constant algebras;

(c) a transformation matrix to a basis compatible with the composition series, that is, the first basis elements form a basis of the first term of the composition series, the next extend these to a basis for the second term etc.

`CompositionFactors(A)`

Compute the composition factors of a composition series for the algebra A. This function returns the same as the second return value of `CompositionSeries` above, but will often be very much quicker.

`MinimalLeftIdeals(A : `*parameters*`)`

`MinimalRightIdeals(A : `*parameters*`)`

`MinimalIdeals(A : `*parameters*`)`

`Limit` RngIntElt *Default :* ∞

Return the minimal left/right/two-sided ideals of A (in non-decreasing size). If `Limit` is set to n, at most n ideals are calculated and the second return value indicates whether all of the ideals were computed.

`MaximalLeftIdeals(A : `*parameters*`)`

`MaximalRightIdeals(A : `*parameters*`)`

`MaximalIdeals(A : `*parameters*`)`

`Limit` RngIntElt *Default :* ∞

Return the maximal left/right/two-sided ideals of A (in non-decreasing size). If `Limit` is set to n, at most n ideals are calculated and the second return value indicates whether all of the ideals were computed.

`JacobsonRadical(A)`

Construct the Jacobson (or nilpotent) radical of A, that is, the intersection of the maximal ideals of A (which is equal to the intersection of the maximal left or right ideals).

```
IsSemisimple(A)
```

Return `true` if the Jacobson radical of A is trivial; otherwise `false`.

```
IsSimple(A)
```

Return `true` if A has no non-trivial composition factor; otherwise `false`.

Example H79E1

We create a division algebra of dimension 4 over the rational field.

```
> Q := MatrixAlgebra< Rationals(), 4 |
>    [0,1,0,0, -1,0,0,0, 0,0,0,-1, 0,0,1,0],
>    [0,0,1,0, 0,0,0,1, -1,0,0,0, 0,-1,0,0]>;
> i := Q.1;
> j := Q.2;
> k := i*j;
> Dimension(Q);
4
> MinimalPolynomial( (1+i+j+k)/2 );
$.1^2 - $.1 + 1
```

Hence, the element `(1+i+j+k)/2` is integral. In fact, together with `1`, `i` and `j` it forms a **Z**-basis of a maximal order in `Q`. We create this maximal order as a structure constant algebra over the integers.

```
> a := [ Q!1, i, j, (1+i+j+k)/2 ];
> T := MatrixAlgebra(Rationals(),4) ! &cat[ Coordinates(Q,a[i]) : i in [1..4] ];
> V := RSpace(Rationals(), 4);
> C := [ V ! Coordinates(Q, a[i]*a[j]) * T^-1 : j in [1..4], i in [1..4] ];
> A := ChangeRing( Algebra< V | C >, Integers() );
> IsAssociative(A);
true
> AA := AssociativeAlgebra(A);
> AA;
Associative Algebra of dimension 4 with base ring Integer Ring
> MinimalPolynomial(AA.4);
$.1^2 - $.1 + 1
```

The so constructed maximal order is ramified at 2 and ∞, hence it should be simple after reducing at odd primes.

```
> for p in [ i : i in [1..100] | IsPrime(i) ] do
>    if not IsSimple( ChangeRing( AA, GF(p) ) ) then
>       print p;
>    end if;
> end for;
2
> CS, CF, T := CompositionSeries( ChangeRing( AA, GF(2) ) );
> T;
[1 0 1 0]
```

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

A glance at the preimages of the basis of the irreducible submodule shows the ramification of AA at the prime 2.

```
> MinimalPolynomial(AA.1 + AA.3);
$.1^2 - 2*$.1 + 2
> MinimalPolynomial(AA.2 + AA.3);
$.1^2 + 2
```

79.4.5 Operations on Subalgebras

`IsZero(A)`

Returns `true` if the algebra A is trivial; otherwise `false`.

`A eq B`

Returns `true` if the algebras A and B (having a common superalgebra) are equal; otherwise `false`.

`A ne B`

Returns `true` if the algebras A and B are not equal; otherwise `false`.

`A subset B`

Returns `true` if A is a subalgebra of the algebra B; otherwise `false`.

`A notsubset B`

Returns `true` if A is not a subalgebra of the algebra B; otherwise `false`.

`A meet B`

The intersection of the algebras A and B, which must have a common superalgebra.

`A * B`

The algebra product $A * B$ of the algebras A and B, which must have a common superalgebra.

`A ^ n`

The (left-normed) n-th power of the algebra A, i.e. $((\dots(A * A) * \dots) * A)$.

`Morphism(A, B)`

The map giving the morphism from A to B. Either A is a subalgebra of B, in which case the embedding of A into B is returned, or B is a quotient algebra of A, in which case the natural epimorphism from A onto B is returned.

79.5 Operations on Elements of an Algebra

79.5.1 Operations on Elements

```
a + b
```

The sum of the elements a and b of an algebra A.

```
-a
```

The negation of the algebra element a.

```
a - b
```

The difference of the elements a and b of an algebra A.

```
a * b
```

The product of the elements a and b of an algebra A.

```
a * r
r * a
```

The product of the element a of the algebra A and the ring element $r \in R$, where R is the coefficient ring of A.

```
a / r
```

The product of the element a of the R-algebra A and the ring element $1/r \in R$, where R is a field.

```
a ^ n
```

If n is positive, form the (left-normed) n-th power $((\ldots((a * a) * a)\ldots) * a)$ of a; if the parent algebra A of a has an identity and n is zero, return the identity; if $n < 0$ and a has an inverse a^{-1} in A, form the n-th power of a^{-1}.

```
MinimalPolynomial(a)
```

If R is a field or the integer ring, return the minimal polynomial of the algebra element a.

```
Parent(a)
```

For an element a in an algebra A return A.

79.5.2 Comparisons and Membership

`a eq b`

Returns `true` if the elements a and b of an algebra A are equal; otherwise `false`.

`a ne b`

Returns `true` if the elements a and b of an algebra A are not equal; otherwise `false`.

`a in A`

Returns `true` if a is in the algebra A; otherwise `false`.

`a notin A`

Returns `true` if a is not in the algebra A; otherwise `false`.

79.5.3 Predicates on Elements

`IsZero(a)`

Returns `true` if the algebra element a is zero; otherwise `false`.

`IsOne(a)`

Returns `true` if the algebra element a is the identity; otherwise `false`.

`IsMinusOne(a)`

Returns `true` if the algebra element a is the negative of the identity element; otherwise `false`.

`IsUnit(a)`

Returns `true` if the element a is a unit, plus the inverse; otherwise `false`.

`IsRegular(a)`

Returns `true` if the element a is regular, that is, is not a zero divisor; otherwise `false`.

`IsZeroDivisor(a)`

Returns `true` if the algebra element a is a divisor of zero; otherwise `false`.

`IsIdempotent(a)`

Returns `true` if the element a is an idempotent, i.e. if $a^2 = a$; otherwise `false`.

`IsNilpotent(a)`

Returns `true` if the element a is nilpotent, i.e. if $a^n = 0$ for some $n \geq 0$; otherwise `false`. If `true`, the minimal n such that $a^n = 0$ is returned as a second value.

80 STRUCTURE CONSTANT ALGEBRAS

Chapter 80
STRUCTURE CONSTANT ALGEBRAS

80.1 Introduction

A structure constant algebra A of dimension n over a ring R can be defined in Magma by giving the n^3 *structure constants* $a_{ij}^k \in R (1 \le i, j, k \le n)$ such that, if $e_1, e_2, \ldots, e_n$ is the basis of A, $e_i * e_j = \sum_{k=1}^{n} a_{ij}^k * e_k$. Structure constant algebras may be defined over any unital ring R. However, many operations require that R be a Euclidean domain or even a field.

80.2 Construction of Structure Constant Algebras and Elements

80.2.1 Construction of a Structure Constant Algebra

There are three ways in Magma to specify the structure constants for a structure constant algebra A of dimension n. The first is to give n^3 ring elements, the second to identify A with the module $M = R^n$ and give the products $e_i * e_j$ as elements of M and the third to specify only the non-zero structure constants.

`Algebra< R, n | Q : parameters >`

`Algebra< M | Q : parameters >`

`Rep` MonStgElt *Default : "Dense"*

This function creates the structure constant algebra A over the free module $M = R^n$, with standard basis $e_1, e_2, \ldots, e_n$, and with the structure constants a_{ij}^k being given by the sequence Q. The sequence Q can be of any of the following three forms. Note that in all cases the actual ordering of the structure constants is the same: it is only their division that varies.

(i) A sequence of n sequences of n sequences of length n. The j-th element of the i-th sequence is the sequence $[a_{ij}^1, \ldots, a_{ij}^n]$, or the element $(a_{ij}^1, \ldots, a_{ij}^n)$ of M, giving the coefficients of the product $e_i * e_j$.

(ii) A sequence of n^2 sequences of length n, or n^2 elements of M. Here the coefficients of $e_i * e_j$ are given by position $(i-1)*n+j$ of Q.

(iii) A sequence of n^3 elements of the ring R. Here the sequence elements are the structure constants themselves, with the ordering $a_{11}^1, a_{11}^2, \ldots, a_{11}^n, a_{12}^1, a_{12}^2, \ldots, a_{nn}^n$. So a_{ij}^k lies in position $(i-1)*n^2+(j-1)*n+k$ of Q.

The optional parameter `Rep` can be used to select the internal representation of the structure constants. The possible values for `Rep` are `"Dense"`, `"Sparse"` and `"Partial"`, with the default being `"Dense"`. In the dense format, the n^3 structure constants are stored as n^2 vectors of length n, similarly to (ii) above. This is the best representation if most of the structure constants are non-zero. The sparse format, intended for use when most structure constants are zero, stores the positions and values of the non-zero structure constants. The partial format stores the vectors, but records for efficiency the positions of the non-zero structure constants.

`Algebra< R, n | T :` *parameters* `>`

`Rep` MonStgElt *Default : "Sparse"*

This function creates the structure constant algebra A with standard basis $e_1, e_2, \ldots, e_n$ over R. The sequence T contains quadruples $< i, j, k, a_{ij}^k >$ giving the non-zero structure constants. All other structure constants are defined to be 0.

As above, the optional parameter `Rep` can be used to select the internal representation of the structure constants.

`ChangeBasis(A, B)`

`ChangeBasis(A, B)`

`ChangeBasis(A, B)`

`Rep` MonStgElt *Default : "Dense"*

Create a new structure constant algebra A', isomorphic to A, by recomputing the structure constants with respect to the basis B. The basis B can be specified as a set or sequence of elements of A, a set or sequence of vectors, or a matrix. The second returned value is the isomorphism from A to A'.

As above, the optional parameter `Rep` can be used to select the internal representation of the structure constants. Note that the default is dense representation, regardless of the representation used by A.

80.2.2 Construction of Elements of a Structure Constant Algebra

`elt< A |` $r_1, r_2, \ldots, r_n$ `>`

Given a structure constant algebra A of dimension n over a ring R, and ring elements $r_1, r_2, \ldots, r_n \in R$ construct the element $r_1 * e_1 + r_2 * e_2 + \ldots + r_n * e_n$ of A.

`A ! Q`

Given a structure constant algebra A of dimension n and a sequence $Q = [r_1, r_2, \ldots, r_n]$ of elements of the base ring R of A, construct the element $r_1 * e_1 + r_2 * e_2 + \ldots + r_n * e_n$ of A.

`BasisProduct(A, i, j)`

Return the product of the i-th and j-th basis element of the algebra A.

`BasisProducts(A)`

`Rep` MonStgElt *Default : "Dense"*

Return the products of all basis elements of the algebra A.

The optional parameter `Rep` may be used to specify the format of the result. If `Rep` is set to "Dense", the products are returned as a sequence Q of n sequences of n elements of A, where n is the dimension of A. The element $Q[i][j]$ is the product of the i-th and j-th basis elements.

If `Rep` is set to "Sparse", the products are returned as a sequence Q containing quadruples (i, j, k, a_{ijk}) signifying that the product of the i-th and j-th basis elements is $\sum_{k=1}^{n} a_{ijk} b_k$, where b_k is the k-th basis element and $n = \dim(A)$.

80.3 Operations on Structure Constant Algebras and Elements

80.3.1 Operations on Structure Constant Algebras

`IsCommutative(A)`

Returns `true` if the algebra A is commutative; otherwise `false`.

`IsAssociative(A)`

Returns `true` if the algebra A is associative; otherwise `false`.

Note that for a structure constant algebra of dimension n this requires up to n^3 tests.

`IsLie(A)`

Returns `true` if the algebra A is a Lie algebra; otherwise `false`.

Note that for a structure constant algebra of dimension n this requires about $n^3/3$ tests of the Jacobi identity.

`DirectSum(A, B)`

Construct a structure constant algebra of dimension $n + m$ where n and m are the dimensions of the algebras A and B, respectively. The basis of the new algebra is the concatenation of the bases of A and B and the products $a * b$ where $a \in A$ and $b \in B$ are defined to be 0.

Example H80E1

We define a structure constant algebra which is a Jordan algebra.

```
> M := MatrixAlgebra( GF(3), 2 );
> B := Basis(M);
> C := &cat[Coordinates(M,(B[i]*B[j]+B[j]*B[i])/2) : j in [1..#B], i in [1..#B]];
> A := Algebra< GF(3), #B | C >;
> #A;
81
> IsAssociative(A);
false
> IsLie(A);
false
> IsCommutative(A);
true
```

This is a good start, as one of the defining properties of Jordan algebras is that they are commutative. The other property is that the identity $(x^2 * y) * x = x^2 * (y * x)$ holds for all $x, y \in A$. We check this on a random pair.

```
> x := Random(A); y := Random(A); print (x^2*y)*x - x^2*(y*x);
(0 0 0 0)
```

The algebra is small enough to check this identity on all elements.

```
> forall{<x, y>: x, y in A | (x^2*y)*x eq x^2*(y*x)};
true
```

So the algebra is in fact a Jordan algebra (which was clear by construction). We finally have a look at the structure constants.

```
> BasisProducts(A);
[
    [ (1 0 0 0), (0 2 0 0), (0 0 2 0), (0 0 0 0) ],
    [ (0 2 0 0), (0 0 0 0), (2 0 0 2), (0 2 0 0) ],
    [ (0 0 2 0), (2 0 0 2), (0 0 0 0), (0 0 2 0) ],
    [ (0 0 0 0), (0 2 0 0), (0 0 2 0), (0 0 0 1) ]
]
```

80.3.2 Indexing Elements

`a[i]`

If a is an element of a structure constant algebra A of dimension n and $1 \leq i \leq n$ is a positive integer, then the i-th component of the element a is returned (as an element of the base ring R of A).

`a[i] := r`

Given an element a belonging to a structure constant algebra of dimension n over R, a positive integer $1 \leq i \leq n$ and an element $r \in R$, the i-th component of the element a is redefined to be r.

80.3.3 The Module Structure of a Structure Constant Algebra

```
Module(A)
```

The module R^n underlying the structure constant algebra A.

```
Degree(A)
```

The degree (= dimension) of the module underlying the algebra A.

```
Degree(a)
```

Given an element belonging to the structure constant algebra A of dimension n, return n.

```
ElementToSequence(a)
Eltseq(a)
```

The sequence of coefficients of the structure constant algebra element a.

```
Coordinates(S, a)
```

Let a be an element of a structure constant algebra A and let S be a subalgebra of A containing a. This function returns the coefficients of a with respect to the basis of S.

```
InnerProduct(a, b)
```

The (Euclidean) inner product of the coefficient vectors of a and b, where a and b are elements of some structure constant algebra A.

```
Support(a)
```

The support of the structure constant algebra element a; i.e. the set of indices of the non-zero components of a.

80.3.4 Homomorphisms

```
hom< A -> B | Q >
```

Given a structure constant algebra A of dimension n over R and either a structure constant algebra B over R or a module B over R, construct the homomorphism from A to B specified by Q. The sequence Q may be of the form $[b_1, \ldots, b_n]$, $b_i \in B$, indicating that the i-th basis element of A is mapped to b_1 or of the form $[< a_1, b_1 >, \ldots, < a_n, b_n >]$ indicating that a_i maps to b_i, where the $a_i (1 \leq i \leq n)$ must form a basis of A.

Note that this is in general only a module homomorphism, it is not checked whether it is an algebra homomorphism.

Example H80E2

We construct the real Cayley algebra, which is a non-associative algebra of dimension 8, containing 7 quaternion algebras. If the basis elements are labelled $1, \ldots, 8$ and 1 corresponds to the identity, these quaternion algebras are spanned by $\{1, (n{+}1) \bmod 7{+}2, (n{+}2) \bmod 7{+}2, (n{+}4) \bmod 7{+}4\}$, where $0 \leq n \leq 6$. We first define a function, which, given three indices i, j, k constructs a sequence with the structure constants for the quaternion algebra spanned by $1, i, j, k$ in the quadruple notation.

```
> quat := func<i,j,k | [<1,1,1, 1>, <i,i,1, -1>, <j,j,1, -1>, <k,k,1, -1>,
>   <1,i,i, 1>, <i,1,i, 1>, <1,j,j, 1>, <j,1,j, 1>, <1,k,k, 1>, <k,1,k, 1>,
>   <i,j,k, 1>, <j,i,k, -1>, <j,k,i, 1>, <k,j,i, -1>, <k,i,j, 1>, <i,k,j, -1>]>;
```

We now define the sequence of non-zero structure constants for the Cayley algebra using the function `quat`. Some structure constants are defined more than once and we have to get rid of these when defining the algebra.

```
> con := &cat[quat((n+1) mod 7 +2, (n+2) mod 7 +2, (n+4) mod 7 +2):n in [0..6]];
> C := Algebra< Rationals(), 8 | Setseq(Set(con)) >;
> C;
Algebra of dimension 8 with base ring Rational Field
> IsAssociative(C);
false
> IsAssociative( sub< C | C.1, C.2, C.3, C.5 > );
true
```

The integral elements in this algebra are those where either all coefficients are integral or exactly 4 coefficients lie in $1/2 + \mathbf{Z}$ in positions i_1, i_2, i_3, i_4, such that i_1, i_2, i_3, i_4 are a basis of one of the 7 quaternion algebras or a complement of such a basis. These elements are called the integral Cayley numbers and form a **Z**-algebra. The units in this algebra are the elements with either one entry ± 1 and the others 0 or with 4 entries $\pm 1/2$ and 4 entries 0, where the non-zero entries are in the positions as described above. This gives 240 units and they form (after rescaling with $\sqrt{2}$) the roots in the root lattice of type E_8.

```
> a := (C.1 - C.2 + C.3 - C.5) / 2;
> MinimalPolynomial(a);
$.1^2 - $.1 + 1
> MinimalPolynomial(a^-1);
$.1^2 - $.1 + 1
> MinimalPolynomial(C.2+C.3);
$.1^2 + 2
> MinimalPolynomial((C.2+C.3)^-1);
$.1^2 + 1/2
```

Tensoring the integral Cayley algebra with a finite field gives a finite Cayley algebra. As the **Z**-algebra generated by the chosen basis for C has index 2^4 in the full integral Cayley algebra, we can get the finite Cayley algebras by applying the `ChangeRing` function for finite fields of odd characteristic. The Cayley algebra over $GF(q)$ has the simple group $G_2(q)$ as its automorphism group. Since the identity has to be fixed, every automorphism is determined by its image on the remaining 7 basis elements. Each of these has minimal polynomial $x^2 + 1$, hence one obtains a permutation representation of $G_2(q)$ on the elements with this minimal polynomial. As ±-pairs have to be preserved, this number can be divided by 2.

```
> C3 := ChangeRing( C, GF(3) );
> f := MinimalPolynomial(C3.2);
> f;
$.1^2 + 1
> #C3;
6561
> time Im := [ c : c in C3 | MinimalPolynomial(c) eq f ];
Time: 3.099
> #Im;
702
> C5 := ChangeRing( C, GF(5) );
> f := MinimalPolynomial(C5.2);
> f;
$.1^2 + 1
> #C5;
390625
> time Im := [ c : c in C5 | MinimalPolynomial(c) eq f ];
Time: 238.620
> #Im;
15750
```

In the case of the Cayley algebra over $GF(3)$ we obtain a permutation representation of degree 351, which is in fact the smallest possible degree (corresponding to the representation on the cosets of the largest maximal subgroup $U_3(3) : 2$). Over $GF(5)$, the permutation representation is of degree 7875, corresponding to the maximal subgroup $L_3(5) : 2$, the smallest possible degree being 3906.

81 ASSOCIATIVE ALGEBRAS

Chapter 81
ASSOCIATIVE ALGEBRAS

81.1 Introduction

Defining an algebra by structure constants gives a very general set-up, but many structural concepts are restricted to associative algebras. Therefore, Magma provides a special type for structure constant algebras which are known to be associative.

81.2 Construction of Associative Algebras

81.2.1 Construction of an Associative Structure Constant Algebra

The construction of an associative structure constant algebra is identical to that of a general structure constant algebra, with the exception that an additional parameter is provided which may be used to avoid checking that the algebra is associative.

`AssociativeAlgebra< R, n | Q : parameters >`

`AssociativeAlgebra< M | Q : parameters >`

`Check`	BoolElt	*Default :* `true`
`Rep`	MonStgElt	*Default :* "*Dense*"

This function creates the associative structure constant algebra A over the free module $M = R^n$, with standard basis $e_1, e_2, \ldots, e_n$, and with the structure constants a_{ij}^k being given by the sequence Q. The sequence Q can be of any of the following three forms. Note that in all cases the actual ordering of the structure constants is the same: it is only their division that varies.

(i) A sequence of n sequences of n sequences of length n. The j-th element of the i-th sequence is the sequence $[a_{ij}^1, \ldots, a_{ij}^n]$, or the element $(a_{ij}^1, \ldots, a_{ij}^n)$ of M, giving the coefficients of the product $e_i * e_j$.

(ii) A sequence of n^2 sequences of length n, or n^2 elements of M. Here the coefficients of $e_i * e_j$ are given by position $(i-1) * n + j$ of Q.

(iii) A sequence of n^3 elements of the ring R. Here the sequence elements are the structure constants themselves, $a_{11}^1, a_{11}^2, \ldots, a_{11}^n, a_{12}^1, a_{12}^2, \ldots, a_{nn}^n$. So a_{ij}^k lies in position $(i-1) * n^2 + (j-1) * n + k$ of Q.

By default the algebra is checked to be associative; this can be overruled by setting the parameter `Check` to `false`.

The optional parameter `Rep` can be used to select the internal representation of the structure constants. The possible values for `Rep` are `"Dense"`, `"Sparse"` and `"Partial"`, with the default being `"Dense"`. In the dense format, the n^3 structure constants are stored as n^2 vectors of length n, similarly to (ii) above. This is the best representation if most of the structure constants are non-zero. The sparse format, intended for use when most structure constants are zero, stores the positions and values of the non-zero structure constants. The partial format stores the vectors, but records for efficiency the positions of the non-zero structure constants.

```
AssociativeAlgebra< R, n | T : parameters >
```

Check	BoolElt	*Default :* true
Rep	MonStgElt	*Default :* "*Sparse*"

This function creates the associative structure constant algebra A with standard basis $e_1, e_2, \ldots, e_n$ over R. The sequence T contains quadruples $< i, j, k, a_{ij}^k >$ giving the non-zero structure constants. All other structure constants are defined to be 0.

The optional parameters are as above.

```
AssociativeAlgebra(A)
```

Given a structure constant algebra A of type `AlgGen`, construct an isomorphic associative structure constant algebra of type `AlgAss`. If it is not known whether or not A is associative, this will be checked and an error occurs if it is not. The elements of the resulting algebra can be coerced into A and vice versa.

```
ChangeBasis(A, B)
ChangeBasis(A, B)
ChangeBasis(A, B)
```

Rep	MonStgElt	*Default :* "*Dense*"

Create a new associative structure constant algebra A', isomorphic to A, by recomputing the structure constants with respect to the basis B. The basis B can be specified as a set or sequence of elements of A, a set or sequence of vectors, or a matrix. The second returned value is the isomorphism from A to A'.

As above, the optional parameter `Rep` can be used to select the internal representation of the structure constants. Note that the default is dense representation, regardless of the representation used by A.

81.2.2 Associative Structure Constant Algebras from other Algebras

```
Algebra(A)
```

If A is either a group algebra of type `AlgGrp` given in vector representation or a matrix algebra of type `AlgMat`, construct the associative structure constant algebra B isomorphic to A together with the isomorphism $A \to B$.

`Algebra(F, E)`

Let E and F be either finite fields or algebraic number fields such that E is a subfield of F. This function returns the associative algebra A of dimension $[F : E]$ over E which is isomorphic to F, together with the isomorphism from F to A such that the $(i-1)$-th power of the generator of F over E is mapped to the i-th basis vector of A.

`AlgebraOverCenter(A)`

Given a simple algebra A of type `AlgMat` or `AlgAss` with center K, this function returns a K-algebra B which is K-isomorphic to A as well as an isomorphism from A to B.

81.3 Operations on Algebras and their Elements

81.3.1 Operations on Algebras

`Centre(A)`

The centre of the associative algebra A.

`Centralizer(A, S)`

`Centraliser(A, S)`

The centralizer of the subalgebra S of the associative algebra A, that is, the subalgebra of A commuting elementwise with S.

`Idealizer(A, B:` *parameters*`)`

`Idealiser(A, B:` *parameters*`)`

`Side` MonStgElt *Default : "Both"*

Given an associative algebra A and a subalgebra B of A, compute the idealizer of B in A, that is, the largest subalgebra of A in which B is an ideal. By default the two-sided idealizer, that is, the largest subalgebra in which B is a two-sided ideal, is found; the left- or right-idealizer can be found by setting the parameter `Side` to `"Left"` or `"Right"` respectively.

`LieAlgebra(A)`

For an associative structure constant algebra A, return the structure constant algebra L with product given by the Lie bracket $(a, b) \mapsto a * b - b * a$. As a second value the map identifying the elements of A and L is returned.

`CommutatorModule(A, B)`

Let A and B be subalgebras of an associative algebra with underlying module M. This function returns the submodule of M which is spanned by the elements $[a, b] = a * b - b * a$, $a \in A, b \in B$.

`CommutatorIdeal(A, B)`

For two subalgebras A and B of an associative algebra, return the ideal generated by all $[a, b] = a * b - b * a$, $a \in A, b \in B$.

`LeftAnnihilator(A, B)`

For two subalgebras A and B of an associative algebra, return the left annihilator of B in A; that is, the subalgebra of A consisting of all elements a such that $a * b = 0$ for all $b \in B$.

`RightAnnihilator(A, B)`

For two subalgebras A and B of an associative algebra, return the right annihilator of B in A; that is, the subalgebra of A consisting of all elements a such that $b * a = 0$ for all $b \in B$.

Example H81E1

We create the Lie algebra $sl_3(\mathbf{Q})$ as a structure constant algebra. First, we construct $gl_3(\mathbf{Q})$ from the full matrix algebra $M_3(\mathbf{Q})$ and get $sl_3(\mathbf{Q})$ as the derived algebra of $gl_3(\mathbf{Q})$.

```
> gl3 := LieAlgebra(Algebra(MatrixRing(Rationals(), 3)));
> sl3 := gl3 * gl3;
> sl3;
Lie Algebra of dimension 8 with base ring Rational Field
```

Let's see how the first basis element acts.

```
> for i in [1..8] do
>     print sl3.i * sl3.1;
> end for;
(0 0 0 0 0 0 0 0)
( 0 -1  0  0  0  0  0  0)
( 0  0 -2  0  0  0  0  0)
(0 0 0 1 0 0 0 0)
(0 0 0 0 0 0 0 0)
( 0  0  0  0  0 -1  0  0)
(0 0 0 0 0 0 2 0)
(0 0 0 0 0 0 0 1)
```

Since it acts diagonally, this element lies in a Cartan subalgebra. The next candidate seems to be the fifth basis element.

```
> for i in [1..8] do
>     print sl3.i * sl3.5;
> end for;
(0 0 0 0 0 0 0 0)
(0 1 0 0 0 0 0 0)
( 0  0 -1  0  0  0  0  0)
( 0  0  0 -1  0  0  0  0)
(0 0 0 0 0 0 0 0)
```

```
( 0  0  0  0  0 -2  0  0)
(0 0 0 0 0 0 1 0)
(0 0 0 0 0 0 0 2)
```

This also acts diagonally and commutes with `sl3.1`, hence we have luckily found a full Cartan algebra in $sl_3(\mathbf{Q})$. We can now easily work out the root system. Obviously the root spaces correspond to the pairs `(sl3.2, sl3.4)`, `(sl3.3, sl3.7)` and `(sl3.6, sl3.8)`. The product of a positive root with its negative should lie in the Cartan algebra.

```
> sl3.2*sl3.4;
( 1  0  0  0 -1  0  0  0)
> sl3.3*sl3.7;
(1 0 0 0 0 0 0 0)
> sl3.6*sl3.8;
(0 0 0 0 1 0 0 0)
```

Clearly some choices have to be made and we fix `sl3.3` as the element e_α corresponding to the first fundamental root α, `sl3.7` as $e_{-\alpha}$ and get `sl3.1` as $h_\alpha = e_\alpha * e_{-\alpha}$. For the other fundamental root β we have to find an element e_β such that $e_\alpha * e_\beta$ is non-zero.

```
> sl3.3*sl3.2;
(0 0 0 0 0 0 0 0)
> sl3.3*sl3.4;
( 0  0  0  0  0 -1  0  0)
> sl3.3*sl3.6;
(0 0 0 0 0 0 0 0)
> sl3.3*sl3.8;
(0 1 0 0 0 0 0 0)
```

We choose `sl3.8` as e_β, `sl3.6` as $e_{-\beta}$ and consequently `-sl3.5` as h_β. This now determines $e_{\alpha+\beta}$ to be `sl3.2` and $e_{-\alpha-\beta}$ to be `sl3.4`.

81.3.2 Operations on Elements

`Centralizer(A, s)`

`Centraliser(A, s)`

The centralizer of the element s of the associative algebra A, that is, the subalgebra of A commuting with s.

`LieBracket(a, b)`

`(a, b)`

The Lie bracket $a * b - b * a$ of a and b, where a and b are elements of an associative algebra A.

`IsScalar(a)`

Returns `true` (and a coerced to F) iff a belongs to the base ring F of its parent algebra.

`RepresentationMatrix(a, M : parameters)`

Side MonStgElt *Default : "Right"*

Returns the matrix representation of Side-multiplication by the element a in the associative algebra A (which must have 1) on the A-module M.

81.3.3 Representations

`MatrixAlgebra(A)`

For an associative algebra A of dimension n return an isomorphic matrix algebra. If A contains the identity-element, the matrix algebra will be of degree n, otherwise it will be of degree $n + 1$.

`MatrixAlgebra(A, M : parameters)`

Side MonStgElt *Default : "Right"*

Given a finite-dimensional R-algebra A and a Side A-module M (both free as R-modules), return the matrix algebra of A-endomorphisms of M, and the R-algebra homomorphism from A into this endomorphism ring.

`RegularRepresentation(A : parameters)`

Side MonStgElt *Default : "Right"*

For an associative algebra A of dimension n over R return its regular representation. If $B = (e_1, e_2, \ldots, e_n)$ is the stored basis for A, an element $a \in A$ is mapped to the matrix in $R^{n \times n}$ which has as its i-th row the coordinates of $e_i * a$ with respect to B. As a second map, the homomorphism of A onto the regular representation is returned.

By default, the right-regular representation is computed. This can be changed to the left-regular representation (in which the i-th row of the image of a contains the coordinates of $a * e_i$) by setting the parameter Side to "Left".

81.3.4 Decomposition of an Algebra

This section describes a few functions that can be used to obtain information on the structure of a finite-dimensional associative algebra.

`JacobsonRadical(A)`

Al MonStgElt *Default : "Default"*

This returns the largest nilpotent ideal of A. This function works for finite-dimensional associative algebras defined over a field of characteristic 0, or over a finite field.

The algorithm used by default is taken from [CIW97]. The meataxe algorithm can be used by setting Al := "Meataxe".

Example H81E2

We compute the Jacobson radical of the group algebra over the field of three elements of a 3-group. In that case it is equal to the augmentation ideal.

```
> G:= SmallGroup( 27, 5 );
> A:= GroupAlgebra( GF(3), G );
> JacobsonRadical( A );
Ideal of dimension 26 of the group algebra A
```

DirectSumDecomposition(A)

IndecomposableSummands(A)

Given an associative algebra A, return the direct sum decomposition of L as a sequence of ideals of L whose sum is L and each of which cannot be further decomposed into a direct sum of ideals. The second sequence return contains the corresponding primitive central idempotents.

For a description of the algorithm we refer to [EG96].

CentralIdempotents(A)

Let Z be the centre of the associative algebra A, and let $J(Z)$ denote its Jacobson radical. This function returns a sequence of primitive orthogonal idempotents in Z such that their images in $Z/J(Z)$ span $J(Z)$. Each such idempotent generates a two-sided ideal in A. The second return value is the sequence of these ideals.

In particular, if A is a semisimple algebra, then this function returns a sequence of primitive orthogonal idempotents spanning Z. Furthermore, the ideals in the second sequence returned are simple algebras, and their direct sum equals A.

For a description of the algorithm we refer to [EG96].

Example H81E3

We compute the direct sum decomposition of a group algebra.

```
> G:= SmallGroup( 10, 2 );
> A:= GroupAlgebra( Rationals(), G );
> ee, II:= CentralIdempotents( A );
> ee[1];
1/10*Id(G) + 1/10*G.2 + 1/10*G.2^2 + 1/10*G.2^3 + 1/10*G.2^4 + 1/10*G.1 +
1/10*G.1 * G.2 + 1/10*G.1 * G.2^2 + 1/10*G.1 * G.2^3 + 1/10*G.1 * G.2^4
> II;
[
    Ideal of dimension 1 of the group algebra A
    Basis:
        Id(G) + G.2 + G.2^2 + G.2^3 + G.2^4 + G.1 + G.1 * G.2 + G.1 * G.2^2 +
        G.1 * G.2^3 + G.1 * G.2^4,
    Ideal of dimension 1 of the group algebra A
    Basis:
        Id(G) + G.2 + G.2^2 + G.2^3 + G.2^4 - G.1 - G.1 * G.2 - G.1 * G.2^2 -
        G.1 * G.2^3 - G.1 * G.2^4,
    Ideal of dimension 4 of the group algebra A
    Basis:
        Id(G) - G.2^4 + G.1 - G.1 * G.2^4
        G.2 - G.2^4 + G.1 * G.2 - G.1 * G.2^4
        G.2^2 - G.2^4 + G.1 * G.2^2 - G.1 * G.2^4
        G.2^3 - G.2^4 + G.1 * G.2^3 - G.1 * G.2^4,
    Ideal of dimension 4 of the group algebra A
    Basis:
        Id(G) - G.2^4 - G.1 + G.1 * G.2^4
        G.2 - G.2^4 - G.1 * G.2 + G.1 * G.2^4
        G.2^2 - G.2^4 - G.1 * G.2^2 + G.1 * G.2^4
        G.2^3 - G.2^4 - G.1 * G.2^3 + G.1 * G.2^4
]
```

We see that here the group algebra is the direct sum of two 1-dimensional and two 4-dimensional ideals. The first idempotent is the sum over all group elements divided by the group order.

81.4 Orders

Let F be a number field with ring of integers R, and let A be a associative algebra over F (finite-dimensional, with 1). An *associative order* O of A is a subring $O \subset A$ which is a projective R-module such that $O \cdot F = A$. We will also refer to an associative order simply as an *order*.

In Magma, associative orders have the type `AlgAssVOrd`, and may be declared for any associative algebra of type `AlgAssV`, namely, `AlgAss`, `AlgMat`, `AlgQuat` and `AlgGrp`. Orders have ideals of type `AlgAssVOrdIdl`, and elements of type `AlgAssVOrdElt`. In the special case where A is a quaternion algebra over the rationals, A has type `AlgQuat` and orders in A have type `AlgQuatOrd`.

Orders, like modules over Dedekind domains, are represented by a pseudobasis, see Section 55.10. Currently, only basic arithmetic functions and procedures are available for general associative orders. Most of the nontrivial functionality currently available is designed for orders in quaternion algebras (over the rationals or number fields). The specialised functions for quaternionic orders are described in Chapter 86.

IMPORTANT WARNING for algebras over the rationals: In Magma, the rationals are *not* considered to be a number field (the type `FldRat` is not a subtype of `FldNum`). Currently, much of the functionality here is designed primarily for algebras whose base field is a `FldNum` (while some of it, but not all, also works for algebras over the `FldRat`). To compute with algebras over **Q**, in many cases the best solution is to create **Q** as a number field at the outset, using `RationalsAsNumberField()`, and create the algebra over this field instead of `Rationals()`.

81.4.1 Creation of Orders

```
Order(R, S)
```

Given a ring R and sequence S of elements of an associative algebra A, returns the order of A generated freely over R by the sequence S. The ring R must be a number ring or $\mathbf{Z}$.

```
Order(S)
```

Given a sequence of elements S of an associative algebra A, returns the order of A generated by the sequence S. The algebra A must be defined over a number field F.

```
Order(S, I)
```

Given a sequence of elements S of an associative algebra A and a sequence I of ideals of a number ring R, returns the order of A generated by the sequence S with coefficient ideals I. The algebra A must be defined over a number field F and have ring of integers R.

```
Order(A, m, I)
```

Given an associative algebra A, a matrix m, and a sequence I of ideals of a number ring R, returns the order of A generated by the sequence of elements specified by the rows of m in the basis of A with coefficient ideals I. The algebra A must be defined over a number field F and have ring of integers R.

```
Order(A, pm)
```

Given an associative algebra A and a pseudomatrix pm, returns the order of A specified by the pseudomatrix pm. The basis of the order is specified by the rows of pm which have coefficients with respect to the basis of A. The algebra A must be defined over a number field with ring of integers R which is the base ring of the pseudomatrix pm.

Example H81E4

We begin by illustrating three methods for creating an associative order.

```
> P<x> := PolynomialRing(Rationals());
> F<b> := NumberField(x^3-3*x-1);
> Z_F := MaximalOrder(F);
> A<alpha,beta,alphabeta> := QuaternionAlgebra<F | -3,b>;
```

First type of constructor takes an algebra, a matrix representing the basis elements, and coefficient ideals.

```
> M := MatrixAlgebra(F,4) ! 1;
> I := [ideal<Z_F | 1> : i in [1..4]];
> O := Order(A, M, I);
> O;
Order of Quaternion Algebra with base ring Field of Fractions of Z_F
```

```
with coefficient ring Maximal Equation Order with defining polynomial x^3 - 3*x
    - 1 over its ground order
```

The second type takes an algebra and a pseudomatrix.

```
> P := PseudoMatrix(I, M);
> O := Order(A, P);
> O;
Order of Quaternion Algebra with base ring Field of Fractions of Z_F
with coefficient ring Maximal Equation Order with defining polynomial x^3 - 3*x
    - 1 over its ground order
```

The third takes simply a sequence of elements.

```
> O := Order([alpha,beta]);
> O;
Order of Quaternion Algebra with base ring Field of Fractions of Z_F
with coefficient ring Maximal Equation Order with defining polynomial x^3 - 3*x
    - 1 over its ground order
```

Example H81E5

Here we give two other examples of order creation.

```
> F<w> := CyclotomicField(3);
> A := FPAlgebra<F, x,y | x^3-3, y^3+5, y*x-w*x*y>;
> Aass, f := Algebra(A);
> Aass;
Associative Algebra of dimension 9 with base ring F
> f;
Mapping from: AlgFP: A to AlgAss: Aass
> S := [f(A.i) : i in [1..2]];
> S;
[ (0 0 1 0 0 0 0 0 0), (0 1 0 0 0 0 0 0 0) ]
> O := Order(S);
> O;
Order of Associative Algebra of dimension 9 with base ring Field of Fractions of R
with coefficient ring Maximal Equation Order with defining polynomial x^2 + x +
    1 over its ground order
>
> A := GroupAlgebra(F, DihedralGroup(6));
> Aass := Algebra(A);
> O := Order([g : g in Generators(Aass)]);
> O;
Order of Associative Algebra of dimension 12 with base ring Field of Fractions
of R with coefficient ring Maximal Equation Order with defining polynomial
x^2 + x + 1 over its ground order
```

```
MaximalOrder(A)
```

Computes a maximal **Z**-order in the semisimple associative algebra A, which must be defined over the rational numbers. The algorithm can be found in [Fri00], §3.5. We refer to [IR93] for a very similar approach.

Example H81E6

First we define two 9 × 9-matrices that generate a 9-dimensional associative algebra. We check that its Jacobson radical is zero, and then we compute a maximal order.

```
> a1 := Matrix( [
> [-184174/80137, -325/80137, 71/2163699, 0, 0, 0, 0, 0, 0],
> [17713719/80137, 92087/80137, -325/80137, 0, 0, 0, 0, 0, 0],
> [-2189265975/80137, -16429806/80137, 92087/80137, 0, 0, 0, 0, 0, 0],
> [0, 0, 0, 64850/80137, 1472/240411, -25/2163699, 0, 0, 0],
> [0, 0, 0, -6237225/80137, -32425/80137, 1472/240411, 0, 0, 0],
> [0, 0, 0, 3305230272/80137, 45310743/80137, -32425/80137, 0, 0, 0],
> [0, 0, 0, 0, 0, 0, 119324/80137, -497/240411, -46/2163699],
> [0, 0, 0, 0, 0, 0, -11476494/80137, -59662/80137, -497/240411],
> [0, 0, 0, 0, 0, 0, -1115964297/80137, -28880937/80137, -59662/80137] ] );
> a2:= Matrix( [
> [0, 0, 0, 282469/240411, 956/2163699, -26/2163699, 0, 0, 0],
> [0, 0, 0, -6486714/80137, -21029/240411, 956/2163699, 0, 0, 0],
> [0, 0, 0, 238511484/80137, -2766918/80137, -21029/240411, 0, 0, 0],
> [0, 0, 0, 0, 0, 0, -85879/240411, -4894/2163699, 64/6491097],
> [0, 0, 0, 0, 0, 0, 5322432/80137, 163145/240411, -4894/2163699],
> [0, 0, 0, 0, 0, 0, -1220999166/80137, -13720122/80137, 163145/240411],
> [-1183167/80137, -11814/80137, -14/80137, 0, 0, 0, 0, 0, 0],
> [-94306842/80137, -2653965/80137, -11814/80137, 0, 0, 0, 0, 0, 0],
> [-79581502242/80137, -1335450240/80137, -2653965/80137, 0, 0, 0, 0, 0, 0] ] );
> M := MatrixAlgebra( Rationals(), 9 );
> A := sub< M | [ a1, a2 ] >;
> Dimension(A);
9
> JacobsonRadical( A );
Matrix Algebra [ideal of A] of degree 9 and dimension 0 with 0 generators over
Rational Field
> O := MaximalOrder( A );
> Discriminant( O );
1
> T :=MultiplicationTable(O);
> T[3][7];
[ -16583482050411285785256, 5672389828626293786946, 1059868937213366777403,
55245368126632733561175, -41598423838438078787076, 1726223870812049536260,
66694491159819102489072, 76373181201401217517416, -114928189655490866071212 ]
```

81.4.2 Attributes

`BaseRing(O)`

`CoefficientRing(O)`

The base ring of the associative order O.

`Algebra(O)`

The container algebra of the associative order O.

`Degree(O)`

`Dimension(O)`

Returns the dimension (or degree) of the order O, equivalently the dimension of its parent algebra as a vector space over its ground field.

`Discriminant(O)`

Returns the discriminant of the order O. If O is a quaternion order, returns the reduced discriminant which is the square root of the usual discriminant.

`FactoredDiscriminant(O)`

Returns the factorization of the discriminant of the order O. If O is a quaternion order, returns the factorization of the reduced discriminant which is the square root of the usual discriminant.

`MultiplicationTable(O)`

Returns the multiplication table of the maximal order O. This is a three dimensional table of structure constants. If T denotes this table, then $T[i][j]$ is a sequence of integers containing the coefficients of the product of the i-th and j-th basis elements with respect to the basis of the order.

`Module(O)`

Return the pseudo matrix describing the basis of the associative order O over a number ring.

`TraceZeroSubspace(O)`

Given an order O in a quaternion algebra, this computes the submodule of elements with trace 0. A basis or a pseudo-basis for this submodule is returned, depending whether the base field of the quaternion algebra is $\mathbf{Q}$ or a number field. (The base ring of O is, respectively, either $\mathbf{Z}$ or an order in that number field.)

81.4.3 Bases of Orders

`Basis(O)`

Returns a basis of the order O. All other elements of the order are integral linear combinations of elements of this basis. Note that the elements of a basis will only be elements of the parent algebra A and may not be elements of O because of the existence of coefficient ideals.

`PseudoBasis(O)`

Returns the pseudobasis of the associative order O over a number ring.

`PseudoMatrix(O)`

Returns the pseudomatrix describing the pseudobasis of the associative order O over a number ring.

`ZBasis(O)`

Returns a **Z**-basis for the order O.

`Generators(O)`

Returns a sequence of generators of O as a module over its base ring.

Example H81E7

We compute an order and show how a basis and a pseudobasis can differ.

```
> P<x> := PolynomialRing(Rationals());
> F<b> := NumberField(x^3-3*x-1);
> Z_F := MaximalOrder(F);
> A := QuaternionAlgebra<F | -3,b>;
> O := Order([1/3*A.1, A.2], [ideal<Z_F | b^2+b+1>, ideal<Z_F | 1>]);
> O;
Order of Quaternion Algebra with base ring Field of Fractions of Z_F
with coefficient ring Maximal Equation Order with defining polynomial x^3 - 3*x
    - 1 over its ground order
> Basis(O);
[ Z_F.1, i, j, k ]
> PseudoBasis(O);
[
    <Principal Ideal of Z_F
    Generator:
        Z_F.1, Z_F.1>,
    <Fractional Principal Ideal of Z_F
    Generator:
        1/3*Z_F.1 + 1/3*Z_F.2 + 1/3*Z_F.3, i>,
    <Principal Ideal of Z_F
    Generator:
        Z_F.1, j>,
    <Fractional Principal Ideal of Z_F
    Generator:
        1/3*Z_F.1 + 1/3*Z_F.2 + 1/3*Z_F.3, k>
]
> PseudoMatrix(O);
Pseudo-matrix over Maximal Equation Order with defining polynomial x^3 - 3*x
     - 1 over its ground order
Principal Ideal of Z_F
Generator:
    Z_F.1 * ( Z_F.1 0 0 0 )
Fractional Principal Ideal of Z_F
Generator:
    1/3*Z_F.1 + 1/3*Z_F.2 + 1/3*Z_F.3 * ( 0 Z_F.1 0 0 )
Principal Ideal of Z_F
Generator:
    Z_F.1 * ( 0 0 Z_F.1 0 )
Fractional Principal Ideal of Z_F
Generator:
    1/3*Z_F.1 + 1/3*Z_F.2 + 1/3*Z_F.3 * ( 0 0 0 Z_F.1 )
> ZBasis(O);
[ Z_F.1, Z_F.2, Z_F.3, (1/3*Z_F.1 + 1/3*Z_F.2 + 1/3*Z_F.3)*i, (1/3*Z_F.1 +
4/3*Z_F.2 + 1/3*Z_F.3)*i, (1/3*Z_F.1 + 4/3*Z_F.2 + 4/3*Z_F.3)*i, j, Z_F.2*j,
Z_F.3*j, (1/3*Z_F.1 + 1/3*Z_F.2 + 1/3*Z_F.3)*k, (1/3*Z_F.1 + 4/3*Z_F.2 +
1/3*Z_F.3)*k, (1/3*Z_F.1 + 4/3*Z_F.2 + 4/3*Z_F.3)*k ]
```

Note that the basis of O does not generate O—one needs to include the coefficient ideals.

81.4.4 Predicates

```
O1 eq O2
```

Return `true` if and only if the orders O_1 and O_2 are equal as subrings of the same algebra.

```
x in O
x notin O
```

Return `true` (respectively, `false`) if the element x of an associative algebra is in the associative order O.

81.4.5 Operations with Orders

```
Adjoin(O, x)
Adjoin(O, x, I)
```

Returns the order obtained by adjoining the element x to the order O, optionally with coefficient ideal I.

```
O1 + O2
```

Returns the sum of the orders O_1 and O_2.

```
O1 meet O2
```

Returns the intersection of the orders O_1 and O_2.

```
O ^ x
```

Returns the conjugate order $x^{-1}Ox$.

Example H81E8

```
> P<x> := PolynomialRing(Rationals());
> F<b> := NumberField(x^3-3*x-1);
> Z_F := MaximalOrder(F);
> F := FieldOfFractions(Z_F);
> A<alpha,beta,alphabeta> := QuaternionAlgebra<F | -3,b>;
> O := Order([alpha,beta]);
> O1 := Order([1/3*alpha,beta], [ideal<Z_F | b^2+b+1>, ideal<Z_F | 1>]);
> Discriminant(O1);
Principal Ideal of Z_F
Generator:
    4/1*F.1 + 12/1*F.2 + 8/1*F.3
> xi := (1 + alpha + (7+5*b+6*b^2)*beta + (3+b+6*b^2)*alphabeta)/2;
> zeta := (-6-25*b-5*b^2)*alpha - 3*beta;
> O2 := Adjoin(O, xi);
> O := O1+O2;
> Discriminant(O);
Ideal of Z_F
Basis:
[2 0 4]
[0 2 4]
[0 0 6]
```

81.5 Elements of Orders

81.5.1 Creation of Elements

```
O ! 0
```

```
Zero(O)
```

The zero element of the associative order O.

```
O ! 1
```

```
One(O)
```

The identity element of the associative order O.

```
O . i
```

Given an associative order O and an integer i, returns the ith basis element as an order over the base ring. Note that the element 1 may or may not be the first element of a basis. These basis elements are returned as elements of the algebra of O not as elements of O itself.

```
O ! x
```

Return an element of the associative order O described by x, where x may be a sequence, an element of an associative order, an element coercible into the coefficient ring of O or into the algebra of O.

```
Random(O)
```

Returns a "random" element of the associative order O with small coefficients.

81.5.2 Arithmetic of Elements

```
x + y
```

The sum of elements x and y of an order of an associative algebra.

```
x - y
```

The difference of elements x and y of an order of an associative algebra.

```
-x
```

The negation of element x of an order of an associative algebra.

```
x * y
```

The product of elements x and y of an associative algebra.

```
u * c
```

```
c * u
```

The product of the element u of an associative order by the scalar c.

`x / y`

The quotient of x by the unit y in the parent algebra.

`x div y`

The exact division of x by y in the order containing them.

`x ^ n`

The product of the element x of an associative order with itself n times.

81.5.3 Predicates on Elements

`x eq y`

Returns `true` if and only if the elements x and y are equal.

`x ne y`

Returns `true` if and only if the elements x and y are not equal.

`IsZero(x)`

Return `true` if the element x of an associative order is the zero element.

`IsUnit(a)`

Return `true` if the element x of an associative order is a unit in that order.

`IsScalar(x)`

Returns `true` if and only if x is an element of the base ring of the order containing it, and if so returns the coerced element.

81.5.4 Other Operations with Elements

`ElementToSequence(x)`

`Eltseq(x)`

Given an element x of an associative order O, returns the sequence of coordinates of x in terms of the basis of O.

`Norm(x)`

The norm of the element x of an order as an element of its parent algebra.

`Trace(x)`

The trace of the element x of an order as an element of its parent algebra.

`LeftRepresentationMatrix(e)`

`RightRepresentationMatrix(e)`

The representation matrix describing left (right) multiplication by the element e of an associative order.

`RepresentationMatrix(a)`

`Side` MonStgElt *Default : "Left"*

The representation matrix of the element a of an associative order. This describes left multiplication unless the parameter `Side` is set to `"Right"`.

`CharacteristicPolynomial(x)`

The characteristic polynomial of the element x of an order as an element of its parent algebra.

`MinimalPolynomial(x)`

The minimal polynomial of the element x of an order as an element of its parent algebra.

81.6 Ideals of Orders

81.6.1 Creation of Ideals

`lideal< O | E >`

`rideal< O | E >`

`ideal< O | E >`

For an associative order O, this constructs the left, right or two sided O-ideal generated by the elements in the given sequence E (these elements should be coercible into O).

`lideal< O | M >`

`rideal< O | M >`

`ideal< O | M >`

Constructs a left, right or two sided ideal of the associative order O whose basis is given by M, which may be either a matrix or a pseudo matrix.

`O * e`

`e * O`

The principal left (right) ideal of the associative order O generated by the element e.

`RandomRightIdeal(O)`

Returns a "random" right ideal of the order O, generated by elements with small coefficients.

81.6.2 Attributes of Ideals

`Algebra(I)`

The container algebra of the associative ideal I.

`Order(I)`

The associative order the associative ideal I was created as an ideal of.

`LeftOrder(I)`

`RightOrder(I)`

The order which maps the associative ideal I to itself under left (right) multiplication.

`Basis(I)`

`Basis(I, R)`

The basis of the associative ideal I. This will be returned as elements of the order or algebra R if this second argument is given, otherwise as elements of the algebra of I.

`BasisMatrix(I)`

`BasisMatrix(I, R)`

The basis matrix of the associative ideal I. This will be with respect to the basis of the order or algebra R if this second argument is given, otherwise with respect to the basis of the order I was created as an ideal of.

`PseudoBasis(I)`

`PseudoBasis(I, R)`

Return a sequence of tuples of the coefficient ideals and the basis elements of the associative ideal I. If a second argument is given, an order or algebra R, then the basis elements will be in R, otherwise the algebra of I.

`PseudoMatrix(I)`

`PseudoMatrix(I, R)`

Return a pseudo matrix describing the basis of the associative ideal I. If a second argument is given, an order or algebra R, then the basis matrix will be with respect to the basis of R, otherwise the order I was created as an ideal of.

`ZBasis(I)`

Returns a **Z**-basis for the ideal I.

`Generators(I)`

Returns a sequence of generators for the ideal I as a module over its base ring.

`Denominator(I)`

Return the denominator of the ideal I. This is the minimal element d of the coefficient ring of O such that $d * I \subseteq O$ where O is the order I was created as an ideal of.

81.6.3 Arithmetic for Ideals

`I + J`

The sum of the ideals I and J, which are ideals which share a side in equal orders.

`I * J`

The product of the ideals I and J, where I is a right ideal and J is a left ideal of the same order O. Returns the product given the structure of left and right ideal.

`a * I`

`I * a`

Returns the product of a and I as an ideal.

`Colon(J, I)`

If I, J are left ideals, returns the colon $(J : I) = \{x \in A : xI \subset J\}$, similarly defined if I, J are right ideals.

`MultiplicatorRing(I)`

Returns the colon $(I : I)$ of the ideal I, the set of all elements which multiply I into I.

81.6.4 Predicates on Ideals

`IsLeftIdeal(I)`

`IsRightIdeal(I)`

`IsTwoSidedIdeal(I)`

Return `true` if the associative ideal I is a left, right or two sided ideal (respectively).

`I eq J`

Return `true` if the associative ideals I and J are equal.

`I subset J`

Returns `true` if and only if the ideal I is contained in the ideal J.

`a in I`

`a notin I`

Return `true` (`false`) if the element a of an associative algebra is contained in the associative ideal I.

81.6.5 Other Operations on Ideals

```
Norm(I)
```

Returns the norm of the ideal I, the ideal of the base number ring of I generated by the norms of the elements in I.

Example H81E9

```
> F<w> := CyclotomicField(3);
> R := MaximalOrder(F);
> A := Algebra(FPAlgebra<F, x, y | x^3-3, y^3+5, y*x-w*x*y>);
> O := Order([A.i : i in [1..9]]);
> MinimalPolynomial(O.2);
$.1^3 + 5/1*R.1
> I := rideal<O | O.2>;
> IsLeftIdeal(I), IsRightIdeal(I), IsTwoSidedIdeal(I);
false true false
> MultiplicatorRing(I) eq O;
true
> PseudoBasis(I);
[
    <Principal Ideal of R
    Generator:
        R.1, (0 R.1 0 0 0 0 0 0 0)>,
    <Principal Ideal of R
    Generator:
        R.1, (0 0 0 R.1 0 0 0 0 0)>,
    <Principal Ideal of R
    Generator:
        R.1, (0 0 0 0 -R.1 - R.2 0 0 0 0)>,
    <Principal Ideal of R
    Generator:
        R.1, (-5/1*R.1 0 0 0 0 0 0 0 0)>,
    <Principal Ideal of R
    Generator:
        R.1, (0 0 0 0 0 0 -R.1 - R.2 0 0)>,
    <Principal Ideal of R
    Generator:
        R.1, (0 0 0 0 0 0 0 R.2 0)>,
    <Principal Ideal of R
    Generator:
        R.1, (0 0 5/1*R.1 + 5/1*R.2 0 0 0 0 0 0)>,
    <Principal Ideal of R
    Generator:
        R.1, (0 0 0 0 0 0 0 0 R.2)>,
    <Principal Ideal of R
    Generator:
        R.1, (0 0 0 0 0 -5/1*R.2 0 0 0)>
]
> ZBasis(I);
[ [0 R.1 0 0 0 0 0 0 0 0], [0 R.2 0 0 0 0 0 0 0 0], [0 0 0 R.1 0 0 0 0 0 0], [0 0 0
    R.2 0 0 0 0 0 0], [0 0 0 0 -R.1 - R.2 0 0 0 0 0], [0 0 0 0 R.1 0 0 0 0 0],
    [-5/1*R.1 0 0 0 0 0 0 0 0 0], [-5/1*R.2 0 0 0 0 0 0 0 0 0] ]
> Norm(I);
Principal Ideal of R
Generator:
    15625/1*R.1
> J := rideal<O | O.3>;
> Norm(J);
Principal Ideal of R
Generator:
    729/1*R.1
> A!1 in I+J;
false
> Denominator(1/6*I);
[1, 0]
> Colon(J,I);
Pseudo-matrix over Maximal Equation Order with defining polynomial x^2 + x + 1
over its ground order
Principal Ideal of R
Generator:
    3/1*R.1 * ( R.1 0 0 0 0 0 0 0 0 0 )
Principal Ideal of R
Generator:
    3/1*R.1 * ( 0 R.1 0 0 0 0 0 0 0 0 )
Principal Ideal of R
Generator:
    R.1 * ( 0 0 R.1 0 0 0 0 0 0 0 )
Fractional Principal Ideal of R
Generator:
    3/5*R.1 * ( 0 0 0 R.1 0 0 0 0 0 0 )
Principal Ideal of R
Generator:
    R.1 * ( 0 0 0 0 R.1 0 0 0 0 0 )
Principal Ideal of R
Generator:
    R.1 * ( 0 0 0 0 0 R.1 0 0 0 0 )
Fractional Principal Ideal of R
Generator:
    -1/5*R.1 * ( 0 0 0 0 0 0 R.1 0 0 0 )
Principal Ideal of R
Generator:
    R.1 * ( 0 0 0 0 0 0 0 R.1 0 0 )
Fractional Principal Ideal of R
Generator:
```

```
1/5*R.1 * ( 0 0 0 0 0 0 0 0 R.1 )
```

81.7 Quaternionic Orders

The following intrinsics which take an argument of type `AlgAssVOrd`, `AlgAssVOrdElt`, or `AlgAssVOrdIdl`, apply only to associative orders of quaternion algebras and are documented in that chapter, Chapter 86.

`MaximalOrder(O)` `pMaximalOrder(O, p)`

`IsMaximal(O)` `IspMaximal(O, p)`

`pMatrixRing(O, p)`

`Embed(Oc, O)`

`LeftIdealClasses(S)` `RightIdealClasses(S)`

`TwoSidedIdealClasses(S)`

`TwoSidedIdealClassGroup(S)`

`OptimizedRepresentation(O)` `OptimisedRepresentation(O)`

`Units(S)` `MultiplicativeGroup(S)` `UnitGroup(S)`

`Conjugate(x)`

`Enumerate(O, A, B)` `Enumerate(O, B)` `Enumerate(I, B)`

`ReducedBasis(O)` `ReducedBasis(I)`

`IsIsomorphic(I, J)` `IsLeftIsomorphic(I, J)`

`IsRightIsomorphic(I, J)` `IsPrincipal(I)`

81.8 Bibliography

[**CIW97**] Arjeh M. Cohen, Gábor Ivanyos, and David B. Wales. Finding the radical of an algebra of linear transformations. *J. Pure Appl. Algebra*, 117/118:177–193, 1997. Algorithms for algebra (Eindhoven, 1996).

[**EG96**] W. Eberly and M. Giesbrecht. Efficient decomposition of associative algebras. In Y. N. Lakshman, editor, *Proceedings of the 1996 International Symposium on Symbolic and Algebraic Computation: ISSAC'96*, pages 170–178, New York, 1996. ACM.

[**Fri00**] Carsten Friedrichs. *Berechnung von Maximalordnungen über Dedekindringen.* Dissertation, Technische Universität Berlin, 2000.
URL:http://www.math.tu-berlin.de/~kant/publications/diss/diss_fried.pdf.gz.

[**IR93**] Gábor Ivanyos and Lajos Rónyai. Finding maximal orders in semisimple algebras over **Q**. *Comput. Complexity*, 3(3):245–261, 1993.

82 FINITELY PRESENTED ALGEBRAS

Chapter 82

FINITELY PRESENTED ALGEBRAS

82.1 Introduction

This chapter describes finitely presented algebras (FPAs) in Magma. An FPA is a quotient of a free associative algebra by an ideal of relations. To compute with these ideals, one constructs *noncommutative* Gröbner bases (GBs), which have many parallels with the standard commutative GBs, discussed in Chapter 105. At the heart of the theory is a noncommutative version of the *Buchberger algorithm* which computes a GB of an ideal of an algebra starting from an arbitrary basis (generating set) of the ideal. One significant difference with the commutative case is that a noncommutative GB may not be finite for a finitely-generated ideal. For overviews of the theory and the basic algorithms, see [Mor94, Li02].

Magma also contains an implementation of a noncommutative generalization of the Faugere F_4 algorithm (due to Allan Steel), based on sparse linear algebra techniques, which usually performs dramatically better than the Buchberger algorithm, and so this is used by Magma by default.

82.2 Representation and Monomial Orders

Let A be the free algebra $K\langle x_1, \ldots, x_n\rangle$ of rank n over a field K. A word in the underlying monoid of A is simply an associative product of the letters (or variables) of A. For consistency with the commutative case, we will call these monoid words *monomials*. Elements of A, called noncommutative polynomials, are finite sums of terms, where a term is the product of a coefficient from K and a monomial. The terms are sorted with respect to an admissible order $<$, which satisfies, for monomials p, q, r, the following conditions:

(a) If $p < q$, then $pr < qr$ and $sp < sq$.

(b) If $p = qr$ then $p > q$ and $p > r$.

Currently Magma only supports the noncommutative graded-lexicographical order (*glex*), which first compares degrees and then uses a left-lexicographical comparison for degree-ties. There is no admissible lexicographic order in the noncommutative case.

82.3 Exterior Algebras

Since V2.15 (December 2008), MAGMA has a special type for *exterior algebras.* Such an algebra is skew-commutative and is a quotient of the free algebra $K\langle x_1, \ldots, x_n\rangle$ by the relations $x_i^2 = 0$ and $x_i x_j = -x_j x_i$ for $1 \leq i, j \leq n$, $i \neq j$. Because of these relations, elements of the algebra can be written in terms of commutative monomials in the variables (via a collection algorithm), and the associated algorithms are much more efficient than for the general noncommutative case. Also, a Gröbner basis of an ideal of an exterior algebra is always finite (in fact, the whole exterior algebra has dimension 2^n as a K-vector space).

Exterior algebras may be constructed with the `ExteriorAlgebra` function below, and all operations applicable to general FP algebras are also applicable to them (so will not be duplicated here). Furthermore, modules over exterior algebras are also allowed: see Chapter 109 for details.

82.4 Creation of Free Algebras and Elements

82.4.1 Creation of Free Algebras

Currently algebras may only be created over fields. Free algebras are objects of type `AlgFr` with elements of type `AlgFrElt`.

```
FreeAlgebra(K, n)
```

Create a free algebra in $n > 0$ variables over the field K. The angle bracket notation can be used to assign names to the indeterminates; e.g., `F<a,b,c> := FreeAlgebra(GF(2), 3);`.

```
ExteriorAlgebra(K, n)
```

Create an exterior algebra in $n > 0$ variables over the field K. The angle bracket notation can be used to assign names to the indeterminates; The angle bracket notation can be used to assign names to the indeterminates; e.g., `F<a,b,c> := ExteriorAlgebra(GF(2), 3);`.

82.4.2 Print Names

The `AssignNames` and `Name` functions can be used to associate names with the indeterminates of free algebras after creation.

```
AssignNames(~F, s)
```

Procedure to change the name of the indeterminates of a free algebra F. The i-th indeterminate will be given the name of the i-th element of the sequence of strings s (for $1 \leq i \leq \#s$); the sequence may have length less than the number of indeterminates of F, in which case the remaining indeterminate names remain unchanged.

This procedure only changes the name used in printing the elements of F. It does *not* assign to identifiers corresponding to the strings the indeterminates in F; to do this, use an assignment statement, or use angle brackets when creating the free algebra.

`Name(F, i)`

Given a free algebra F, return the i-th indeterminate of F (as an element of F).

82.4.3 Creation of Polynomials

The easiest way to create (noncommutative) polynomials in a given algebra is to use the angle bracket construction to attach variables to the indeterminates, and then to use these variables to create polynomials (see the examples). Below we list other options.

`F . i`

Return the i-th indeterminate for the free algebra F in n variables ($1 \leq i \leq n$) as an element of F.

`elt< R | a >`

`R ! s`

`elt< R | s >`

This element constructor can only be used for trivial purposes in noncommutative free algebras: given a free algebra $F = R[x_1, \ldots, x_n]$ and an element a that can be coerced into the coefficient ring R, the constant polynomial a is returned; if a is in F already it will be returned unchanged.

`One(F)` `Identity(F)` `Zero(F)` `Representative(F)`

82.5 Structure Operations

82.5.1 Related Structures

The main structure related to a free algebra is its coefficient ring. Multivariate free algebras belong to the Magma category `AlgFr`.

`BaseRing(F)`

`CoefficientRing(F)`

Return the coefficient ring of the free algebra F.

`Category(F)` `Parent(F)` `PrimeRing(F)`

82.5.2 Numerical Invariants

Note that the # operator only returns a value for finite (quotients of) free algebras.

`Rank(F)`

Return the number of indeterminates of free algebra F over its coefficient ring.

`Characteristic(F)` `# F`

82.5.3 Homomorphisms

In its most general form, a homomorphism taking a free algebra $K\langle x_1, \ldots, x_n\rangle$ as domain requires $n + 1$ pieces of information, namely, a map (homomorphism) telling how to map the coefficient ring K together with the images of the n indeterminates. The map for the coefficient ring is optional.

```
hom< F -> S | f, y1, ..., yn >
```

```
hom< F -> S | y1, ..., yn >
```

Given a free algebra $F = K\langle x_1, \ldots, x_n\rangle$, a ring or associative algebra S (including another FP-algebra or a matrix algebra), and a map $f : K \to S$ and n elements $y_1, \ldots, y_n \in S$, create the homomorphism $g : F \to S$ by applying the rules that $g(rx_1^{a_1} \cdots x_n^{a_n}) = f(r)y_1^{a_1} \cdots y_n^{a_n}$ for monomials and linearity, that is, $g(M + N) = g(M) + g(N)$.

The coefficient ring map may be omitted, in which case the coefficients are mapped into S by the coercion map.

No attempt is made to check whether the map defines a genuine homomorphism.

Example H82E1

In this example we map an algebra F first into F itself, and then into a matrix algebra.

```
> K := RationalField();
> F<x,y,z> := FreeAlgebra(K, 3);
> h := hom<F -> F | x*y, y*x, z*x>;
> h(x);
x*y
> h(y);
y*x
> h(x*y);
x*y^2*x
> h(x + y + z);
x*y + y*x + z*x
> A := MatrixAlgebra(K, 2);
> M := [A | [1,1,-1,1], [-1,3,4,1], [11,7,-7,8]];
> M;
[
    [ 1  1]
    [-1  1],

    [-1  3]
    [ 4  1],

    [11  7]
    [-7  8]
]
> h := hom<F -> A | M>;
> h(x);
```

```
[ 1  1]
[-1  1]
> h(y);
[-1  3]
[ 4  1]
> h(x*y - y*z);
[ 35 -13]
[-32 -38]
```

82.6 Element Operations

82.6.1 Arithmetic Operators

The usual unary and binary ring operations are available for noncommutative polynomials, noting that multiplication is associative but noncommutative, of course.

```
+ a    - a
a + b    a - b    a * b    a ^ k    a / b    a div b
a +:= b    a -:= b    a *:= b    a div:= b
```

82.6.2 Equality and Membership

```
a eq b    a ne b
a in R    a notin R
```

82.6.3 Predicates on Algebra Elements

```
IsZero(f)    IsOne(f)    IsMinusOne(f)
IsNilpotent(f)    IsIdempotent(f)
IsUnit(f)    IsZeroDivisor(f)    IsRegular(f)
IsIrreducible(f)    IsPrime(f)
```

82.6.4 Coefficients, Monomials, Terms and Degree

The functions in this subsection allow one to access noncommutative polynomials.

```
Coefficients(f)
```

Given a noncommutative polynomial f with coefficients in R, this function returns a sequence of 'base' coefficients, that is, a sequence of elements of R occurring as coefficients of the monomials in f. Note that the monomials are ordered, and that the sequence of coefficients corresponds exactly to the sequence of monomials returned by `Monomials(f)`.

```
LeadingCoefficient(f)
```

Given a noncommutative polynomial f with coefficients in R, this function returns the leading coefficient of f as an element of R; this is the coefficient of the leading monomial of f, that is, the first among the monomials occurring in f with respect to the ordering of monomials used in F.

```
TrailingCoefficient(f)
```

Given a noncommutative polynomial f with coefficients in R, this function returns the trailing coefficient of f as an element of R; this is the coefficient of the trailing monomial of f, that is, the last among the monomials occurring in f with respect to the ordering of monomials used in F.

```
MonomialCoefficient(f, m)
```

Given a noncommutative polynomial f and a monomial m, this function returns the coefficient with which m occurs in f as an element of R.

```
Monomials(f)
```

Given a noncommutative polynomial $f \in F$, this function returns a sequence of the monomials (monoid words) occurring in f. Note that the monomials in F are ordered, and that the sequence of monomials corresponds exactly to the sequence of coefficients returned by `Coefficients(f)`.

```
LeadingMonomial(f)
```

Given a noncommutative polynomial $f \in F$ this function returns the leading monomial of f, that is, the first monomial element of F that occurs in f, with respect to the ordering of monomials used in F.

```
Terms(f)
```

Given a noncommutative polynomial $f \in F$, this function returns the sequence of (non-zero) terms of f as elements of F. The terms are ordered according to the ordering on the monomials in F. Consequently the i-th element of this sequence of terms will be equal to the product of the i-th element of the sequence of coefficients and the i-th element of the sequence of monomials.

`LeadingTerm(f)`

Given a noncommutative polynomial $f \in F$, this function returns the leading term of f as an element of F; this is the product of the leading monomial and the leading coefficient that is, the first among the monomial terms occurring in f with respect to the ordering of monomials used in F.

`TrailingTerm(f)`

Given a noncommutative polynomial $f \in F$, this function returns the trailing term of f as an element of F; this is the last among the monomial terms occurring in f with respect to the ordering of monomials used in F.

`Length(m)`

Given a noncommutative monomial (word) m, return the length of m, i.e., the number of letters of m. Note that this differs from the commutative case, where the number of terms in a polynomial is returned.

`m[i]`

Given a noncommutative monomial (word) m of length l, and an integer i with $1 \leq i \leq l$, return the i-th letter of m.

`TotalDegree(f)`

Given a noncommutative polynomial f, this function returns the total degree of f, which is the maximum of the lengths of all monomials that occur in f. If f is the zero polynomial, the return value is -1.

`LeadingTotalDegree(f)`

Given a noncommutative polynomial, this function returns the leading total degree of f, which is the length of the leading monomial of f.

Example H82E2

In this example we illustrate the above access functions.

```
> K := RationalField();
> F<x,y,z> := FreeAlgebra(K, 3);
> f := (3*x*y - 2*y*z)*(4*x - 7*z*y) + 23*x*y*z;
> f;
-21*x*y*z*y + 14*y*z^2*y + 12*x*y*x + 23*x*y*z - 8*y*z*x
> TotalDegree(f);
4
> Coefficients(f);
[ -21, 14, 12, 23, -8 ]
> Monomials(f);
[
    x*y*z*y,
    y*z^2*y,
    x*y*x,
    x*y*z,
    y*z*x
]
> Terms(f);
[
    -21*x*y*z*y,
    14*y*z^2*y,
    12*x*y*x,
    23*x*y*z,
    -8*y*z*x
]
> MonomialCoefficient(f, x*y*z);
23
> LeadingTerm(f);
-21*x*y*z*y
> LeadingCoefficient(f);
-21
> m := Monomials(f)[1];
> m;
x*y*z*y
> Length(m);
4
> m[1];
x
> m[2];
y
```

82.6.5 Evaluation

`Evaluate(f, s)`

Given an element f of a free algebra $F = R\langle x_1, \ldots, x_n\rangle$ and a sequence or tuple s of ring or algebra elements of length n, return the value of f at s, that is, the value obtained by substituting $x_i = s[i]$. This behaves in the same way as the `hom` constructor above.

If the elements of s lie in a ring and can be lifted into the coefficient ring R, then the result will be an element of R. If the elements of s cannot be lifted to the coefficient ring, then an attempt is made to do a generic evaluation of f at s. In this case, the result will be of the same type as the elements of s.

Example H82E3

In this example we illustrate the above access functions.

```
> K := RationalField();
> F<x,y,z> := FreeAlgebra(K, 3);
> g := x*y + y*z;
> g;
x*y + y*z
> Evaluate(g, [1,2,3]);
8
> Parent($1);
Rational Field
> Evaluate(g, [y,x,z]);
x*z + y*x
> Parent($1);
Finitely presented algebra of rank 3 over Rational Field
Non-commutative Graded Lexicographical Order
Variables: x, y, z
```

82.7 Ideals and Gröbner Bases

Magma supports left-sided, right-sided, and two-sided ideals of free algebras. In general, there are not many operations applicable to single-sided ideals: quotients are supported only in the case of two-sided ideals.

Within the general context of fp-algebras, the term “basis” will refer to an *ordered* sequence of polynomials which generate an ideal. (Thus a basis may contain duplicates and zero elements so it is dissimilar to a basis of a vector space.)

82.7.1 Creation of Ideals

`ideal< A | L >`

`lideal< A | L >`

`rideal< A | L >`

Given a free algebra A over a field K, return the two-sided (`ideal`), left-sided (`lideal`), or right-sided (`rideal`) of A generated by the elements of A specified by the list L. Each term of the list L must be an expression defining an object of one of the following types:

(a) An element of A;

(b) A set or sequence of elements of A;

(c) An ideal of A;

(d) A set or sequence of ideals of A.

`Basis(I)`

Given an ideal I, return the current basis of I. If a Gröbner basis of I has been computed, that is returned.

```
BasisElement(I, i)
```

Given an ideal I together with an integer i, return the i-th element of the current basis of I. This the same as `Basis(I)[i]`.

82.7.2 Gröbner Bases

Gröbner bases (GBs) may be computed for any kind of ideal (left-, right-, or two-sided), but for single-sided ideals, the GBs are generally weak (i.e., they rarely differ much from the original generators of the ideals).

Unfortunately, the GB of a given ideal may not be finite. Thus the Buchberger or F_4 algorithms below will run forever in such cases. One can interrupt any GB computation by pressing `Ctrl-C`. Alternatively, the function `GroebnerBasis(S,d)` below, which creates a truncated degree-d Gröbner basis, can be used to set a limit on the degrees of the pairs considered, so the computation will always terminate.

As in the commutative case, when Magma constructs a GB G of an ideal I, then G is always the unique sorted minimal reduced GB of I. Before this happens, an ideal will usually possess a basis which is not a Gröbner basis, but that will be changed into the unique Gröbner basis when the GB is computed. Thus the original basis will be discarded. See the procedure `Groebner` below for details on the algorithms available.

The unique Gröbner basis will be computed automatically when necessary; the `Groebner` procedure below simply allows control of the algorithms used to compute the Gröbner basis.

`Groebner(I:` *parameters*`)`

(Procedure.) Explicitly force a Gröbner basis (GB) for I to be constructed. This procedure is normally not necessary, as Magma will automatically compute the GB when needed, but it does allow one to control how the GB is constructed.

`Faugere` BoolElt *Default :* `true`

Magma has two algorithms for computing noncommutative GBs:

(1) A noncommutative generalization (due to Allan Steel) of the Faugère F_4 algorithm [Fau99], which works by specialized sparse linear algebra and is applicable to two-sided ideals defined over a finite field or the rational field;

(2) The noncommutative Buchberger algorithm [CLO96, Chap. 2, §7] for ideals defined over any field.

If the parameter `Faugere` is set to `true`, then the Faugère F_4 algorithm will be used (if the field is a finite field or the rational field); otherwise the Buchberger algorithm is used.

The current implementation of the Faugère algorithm is usually very much faster than the Buchberger algorithm and usually does not take much more memory, so that it is why it is selected by default. However, there may be examples for which it may be more desirable to use the Buchberger algorithm (particularly to save some memory).

`GroebnerBasis(I: parameters)`

Given an ideal I, force the Gröbner basis of I to be computed, and then return that. The parameters are the same as those for the procedure `Groebner`.

`GroebnerBasis(S: parameters)`

Given a set or sequence S of polynomials, return the unique Gröbner basis of the two-sided ideal generated by S as a sorted sequence. This function is useful for computing Gröbner bases without the need to construct ideals. The parameters are the same as those for the procedure `Groebner`. See also the function `GroebnerBasis(S,d)` below, which creates a truncated degree-d Gröbner basis.

`GroebnerBasis(S, d: parameters)`

Given a set or sequence S of polynomials, return the degree-d Gröbner basis of the ideal generated by S, which is the truncated Gröbner basis obtained by ignoring S-polynomial pairs whose total degree is greater than d.

If the ideal is homogeneous, then it is guaranteed that the result is equal to the set of all polynomials in the full Gröbner basis of the ideal whose total degree is less than or equal to d, and a polynomial whose total degree is less than or equal to d is in the ideal if and only if its normal form with respect to this truncated basis is zero. But if the ideal is not homogeneous, these last properties may not hold, but it may still be useful to construct the truncated basis.

The parameters are the same as those for the procedure `Groebner`.

82.7.3 Verbosity

This subsection describes the verbose flags available for the Gröbner basis algorithms. There are separate verbose flags for each algorithm (`Buchberger`, etc.), but the all-encompassing verbose flag `Groebner` includes all these flags implicitly.

For each procedure provided for setting one of these flags, the value `false` is equivalent to level 0 (nothing), and `true` is equivalent to level 1 (minimal verbosity). For each `Set-` procedure, there is also a corresponding `Get-` function to return the value of the corresponding flag.

`SetVerbose("Groebner", v)`

(Procedure.) Change the verbose printing level for all Gröbner basis algorithms to be v. This includes all of the algorithms whose verbosity is controlled by flags subsequently listed, as well as some other minor related algorithms. Currently the legal levels are 0, 1, 2, 3, or 4. One would normally set this flag to 1 for minimal verbosity for Gröbner basis-type computations, and possibly also set one or more of the following flags to levels higher than 1 for more verbosity.

`SetVerbose("Buchberger", v)`

(Procedure.) Change the verbose printing level for the Buchberger algorithm to be v. Currently the legal levels are 0, 1, 2, 3, or 4. If the value w of the `Groebner` verbose flag is greater than v, then w is taken to be the current value of this flag.

`SetVerbose("Faugere", v)`

(Procedure.) Change the verbose printing level for the Faugère algorithm to be v. Currently the legal levels are 0, 1, 2, or 3. If the value w of the `Groebner` verbose flag is greater than v, then w is taken to be the current value of this flag.

82.7.4 Related Functions

The following functions and procedures perform operations related to Gröbner bases.

`MarkGroebner(I)`

(Procedure.) Given an ideal I, mark the current basis of I to be *the* Gröbner basis of the ideal with respect to the monomial order of the ideal. Note that the current basis must exactly equal the unique (reverse) sorted minimal reduced Gröbner basis for the ideal, as returned by the function `GroebnerBasis`. This procedure is useful when one creates an ideal with a basis known to be the Gröbner basis of the ideal from a previous computation or for other reasons. If the basis is not the unique Gröbner basis, the results are unpredictable.

`Reduce(S)`

Given a set or sequence S of polynomials, return the sequence consisting of the reduction of S. The reduction is obtained by reducing to normal form each element of S with respect to the other elements and sorting the resulting non-zero elements left. Note that all Gröbner bases returned by Magma are automatically reduced so that this function would usually only be used just to simplify a set or sequence of polynomials which is not a Gröbner basis.

Example H82E4

For a certain sequence B of noncommutative polynomials, we create the left-, right- and two-sided ideals generated by B. We note that for the first two cases, the GB is no different from B, but for the two-sided case, the GB contains several more elements.

```
> K := RationalField();
> F<x,y,z> := FreeAlgebra(K, 3);
> B := [x^2 - y*z, x*y - y*z, y*x - z^2, y^3 - x*z];
> I := lideal<F | B>;
> I;
Left ideal of Finitely presented algebra of rank 3 over Rational Field
Non-commutative Graded Lexicographical Order
Variables: x, y, z
Basis:
[
    x^2 - y*z,
    x*y - y*z,
    y*x - z^2,
    y^3 - x*z
]
> GroebnerBasis(I);
[
    y^3 - x*z,
    x^2 - y*z,
    x*y - y*z,
    y*x - z^2
]
> I := rideal<F | B>;
> GroebnerBasis(I);
[
    y^3 - x*z,
    x^2 - y*z,
    x*y - y*z,
    y*x - z^2
]
> I := ideal<F | B>;
> Groebner(I);
> I;
Two-sided ideal of Finitely presented algebra of rank 3 over Rational Field
Non-commutative Graded Lexicographical Order
Variables: x, y, z
Groebner basis:
[
    y*z^2*y - y*z^2,
    y*z^3 - y*z^2,
    z*y*z^2 - y*z^2,
    z^2*y^2 - y*z^2,
    z^2*y*z - y*z^2,
    z^3*y - y*z^2,
    z^4 - y*z^2,
    x*z*x - y*z^2,
    x*z*y - z^3,
    x*z^2 - y*z^2,
    y^3 - x*z,
    y^2*z - z^2*y,
    y*z*x - y*z^2,
    y*z*y - y*z^2,
    z^2*x - z^2*y,
    x^2 - y*z,
    x*y - y*z,
    y*x - z^2
]
> NormalForm(x*y, I);
```

```
y*z
> NormalForm(y*x, I);
z^2
```

Finally, we compute some truncated bases of the two-sided ideal. For degree 2, the truncated GB has no new polynomials while for degree 3, some are added. Only at degree 5 do we obtain the full GB.

```
> GroebnerBasis(B, 2);
[
    y^3 - x*z,
    x^2 - y*z,
    x*y - y*z,
    y*x - z^2
]
> GroebnerBasis(B, 3);
[
    x*z^2 - y*z^2,
    y^3 - x*z,
    y^2*z - z^2*y,
    y*z*x - y*z^2,
    y*z*y - y*z^2,
    z^2*x - z^2*y,
    x^2 - y*z,
    x*y - y*z,
    y*x - z^2
]
> #GroebnerBasis(I);
18
> #GroebnerBasis(B, 4);
16
> #GroebnerBasis(B, 5);
18
> GroebnerBasis(B, 5) eq GroebnerBasis(I);
true
```

82.8 Basic Operations on Ideals

In the following, note that the free algebra F itself is a valid ideal (the ideal containing 1).

82.8.1 Construction of New Ideals

```
I + J
```

Given ideals I and J belonging to the same algebra F, return the sum of I and J, which is the ideal generated by the union of the generators of I and J.

```
I * J
```

Given ideals I and J belonging to the same algebra A, return the product of I and J, which is the ideal generated by the products of the generators of I with those of J.

```
F / J
```

Given an algebra F over a field and an ideal J of F, return the fp-algebra F/J (see below).

```
Generic(I)
```

Given an ideal I of a generic algebra A, return A.

82.8.2 Ideal Predicates

```
I eq J
```

Given two ideals I and J belonging to the same algebra F, return whether I and J are equal.

```
I ne J
```

Given two ideals I and J belonging to the same algebra F, return whether I and J are not equal.

```
I notsubset J
```

Given two ideals I and J belonging to the same algebra F return whether I is not contained in J.

```
I subset J
```

Given two ideals I and J belonging to the same algebra F return whether I is contained in J.

```
IsZero(I)
```

Given an ideal I of the algebra F, return whether I is the zero ideal (contains zero alone).

82.8.3 Operations on Elements of Ideals

```
f in I
```

Given a polynomial f from an algebra F, together with an ideal I of F, return whether f is in I.

```
NormalForm(f, I)
```

Given a polynomial f from an algebra F, together with an ideal I of F, return the unique normal form of f with respect to (the Gröbner basis of) I. The normal form of f is zero if and only if f is in I.

```
NormalForm(f, S)
```

Given a polynomial f from an algebra F, together with a set or sequence S of polynomials from F, return a normal form of f with respect to S. This is not unique in general. If the normal form of f is zero then f is in the ideal generated by S, but the converse is false in general. In fact, the normal form is unique if and only if S forms a Groëbner basis.

```
f notin I
```

Given a polynomial f from an algebra F, together with an ideal I of F, return whether f is not in I.

Example H82E5

We demonstrate the element operations with respect to an ideal of $\mathbf{Q}[x, y, z]$.

```
> F<x,y,z> := FreeAlgebra(RationalField(), 3);
> I := ideal<F | (x + y)^3, (y - z)^2, y^2*z + z>;
> NormalForm(y^2*z + z, I);
0
> NormalForm(x^3, I);
-x^2*y - x*y*x - x*y*z - x*z*y + x*z^2 - y*x^2 - y*x*y - y*z*x -
    y*z*y - z*y*x - z*y*z + z^2*x + z^3
> NormalForm(z^4 + y^2, I);
z^4 + y*z + z*y - z^2
> x + y in I;
false
```

82.9 Changing Coefficient Ring

The `ChangeRing` function enables the changing of the coefficient ring of an algebra or ideal.

```
ChangeRing(I, S)
```

Given an ideal I of an algebra $F = R[x_1, \ldots, x_n]$ of rank n with coefficient ring R, together with a ring S, construct the ideal J of the algebra $Q = S[x_1, \ldots, x_n]$ obtained by coercing the coefficients of the elements of the basis of I into S. It is necessary that all elements of the old coefficient ring R can be automatically coerced into the new coefficient ring S. If R and S are fields and R is known to be a subfield of S and the current basis of I is a Gröbner basis, then the basis of J is marked automatically to be a Gröbner basis of J.

82.10 Finitely Presented Algebras

A finitely presented algebra (fp-algebra) in Magma is simply the quotient ring of a free algebra $F = R\langle x_1, \ldots, x_n\rangle$ by an ideal J of F. It is an object of type `AlgFP` with elements of type `AlgFPElt`.

The elements of fp-algebras are simply noncommutative polynomials which are always kept reduced to normal form modulo the ideal J of “relations”. Practically all operations which are applicable to noncommutative polynomials are also applicable in Magma to elements of fp-algebras (when meaningful).

If an fp-algebra A has finite dimension, considered as a vector space over its coefficient field, then extra special operations are available for A and its elements.

82.11 Creation of FP-Algebras

One can create an fp-algebra simply by forming the quotient of a free algebra by an ideal (`quo` constructor or `/` function). A special constructor `FPAlgebra` is also provided to remove the need to create the free algebra.

NOTE: When one creates an fp-algebra, the ideal of relations is left unchanged, but as soon as one does just about any operation with elements of the algebra (such as printing or multiplying), then the Gröbner basis of the underlying ideal will have to be computed to enable Magma to compute a unique form of each element of the algebra.

```
quo< F | J >
```

```
quo< F | a1, ..., ar >
```

Given a free algebra F and a two-sided ideal J of F, return the fp-algebra (quotient algebra) F/J. The ideal J may either be specified as an ideal or by a list a_1, a_2, ..., a_r, of generators which all lie in F. The angle bracket notation can be used to assign names to the indeterminates: `A<q, r> := quo< I | I.1 + I.2, I.2^2 - 2, I.3^2 + I.4 >;`.

`F / J`

Given a free algebra F and an ideal J of F, return the fp-algebra F/J.

`FPAlgebra<  K, X | L  >`

Given a field K, a list X of n identifiers, and a list L of noncommutative polynomials (relations) in the n variables X, create the fp-algebra of rank n with base ring K with given quotient relations; i.e., return $K[X]/\langle L\rangle$. The angle bracket notation can be used to assign names to the indeterminates.

Example H82E6

We illustrate equivalent ways of creating FP-algebras.

```
> K := RationalField();
> A<x,y> := FPAlgebra<K, x, y | x^2*y - y*x, x*y^3 - y*x>;
> A;
Finitely Presented Algebra of rank 2 over Rational Field
Non-commutative Graded Lexicographical Order
Variables: x, y
Quotient relations:
[
    x^2*y - y*x,
    x*y^3 - y*x
]
> x;
x
> x*y;
x*y
> x^2*y;
y*x
> A;
Finitely Presented Algebra of rank 2 over Rational Field
Non-commutative Graded Lexicographical Order
Variables: x, y
Quotient relations:
[
    x*y*x^2 - y*x^2,
    x*y*x*y - y^2*x,
    x*y^2*x - y^2*x,
    x*y^3 - y*x,
    y*x^3 - y*x^2,
    y*x*y*x - y^2*x,
    y*x*y^2 - x*y*x,
    y^2*x^2 - y^2*x,
    y^2*x*y - y*x^2,
    y^3*x - y*x^2,
    x^2*y - y*x
]
```

The following is equivalent.

```
> K := RationalField();
> F<x,y> := FreeAlgebra(K, 2);
> A<x,y> := quo<F | x^2*y - y*x, x*y^3 - y*x>;
```

82.12 Operations on FP-Algebras

This section describes operations on fp-algebras. Most of the operations are very similar to those for noncommutative free algebras; such operations are done by mapping the computation to the preimage ideal and then by mapping the result back into the fp-algebra. See the corresponding functions for the noncommutative free algebras for details.

`A . i`

Given an fp-algebra A, return the i-th indeterminate of A as an element of A.

`CoefficientRing(A)`

Return the coefficient ring of the fp-algebra A.

`Rank(A)`

Return the rank of the fp-algebra A (the number of indeterminates of A).

`DivisorIdeal(I)`

Given an ideal I of an fp-algebra A which is the quotient ring F/J, where F is a free algebra and J an ideal of F, return the ideal J.

`PreimageIdeal(I)`

Given an ideal I of an fp-algebra A which is the quotient ring F/J, where F is a free algebra and J an ideal of F, return the ideal I' of F such that the image of I' under the natural epimorphism $F \to A$ is I.

`PreimageRing(A)`

Given an fp-algebra A which is the quotient ring F/J, where F is a free algebra and J an ideal of F, return the free algebra F.

`OriginalRing(A)`

Return the generic free algebra F such that A is F/J for some ideal J of F.

`IsCommutative(A)`

Return whether the algebra A is commutative.

```
I eq J
```

Given two ideals I and J of the same fp-algebra A, return `true` if and only if I and J are equal.

```
I subset J
```

Given two ideals I and J of the same fp-algebra A, return `true` if and only if I is contained in J.

```
I + J
```

Given two ideals I and J of the same fp-algebra A, return the sum $I + J$.

```
I * J
```

Given two ideals I and J of the same fp-algebra A, return the product $I * J$.

```
IsProper(I)
```

Given an ideal I of the fp-algebra A, return whether I is proper; that is, whether I is strictly contained in A.

```
IsZero(I)
```

Given an ideal I of the fp-algebra A, return whether I is the zero ideal. Note that this is equivalent to whether the preimage ideal of I is the divisor ideal of A.

82.13 Finite Dimensional FP-Algebras

If an fp-algebra A has finite dimension, considered as a vector space over its coefficient field, then extra special operations are available for A and the elements of A.

```
Dimension(A)
```

Given a finite dimensional fp-algebra A, return the dimension of A.

```
VectorSpace(A)
```

Given a finite dimensional fp-algebra A, construct the vector space V isomorphic to A, and return V together with the isomorphism f from A onto V.

```
MatrixAlgebra(A)
```

Given a finite dimensional fp-algebra A, construct the matrix algebra M isomorphic to A, and return M together with the isomorphism f from A onto M.

```
Algebra(A)
```

Given a finite dimensional fp-algebra A, construct the associative structure-constant algebra S isomorphic to A, and return S together with the isomorphism f from A onto S.

```
RepresentationMatrix(f)
```

Given an element f of a finite dimensional fp-algebra A, return the representation matrix of f, which is a d by d matrix over the coefficient field of A which represents f (where d is the dimension of A).

```
IsUnit(f)
```

Given an element f of a finite dimensional fp-algebra A defined over a field, return whether f is a unit.

```
IsNilpotent(f)
```

Given an element f of a finite dimensional fp-algebra A defined over a field, return whether f is nilpotent, and if so, return also the smallest q such that $f^q = 0$.

```
MinimalPolynomial(f)
```

Given an element f of a finite dimensional fp-algebra A defined over a field, return the minimal polynomial of f as a univariate polynomial over the coefficient field of A.

Example H82E7

We demonstrate the functions available for finite-dimensional fp-algebras.

```
> K := RationalField();
> A<x,y,z> := FPAlgebra<K, x,y,z |
>           x^2 - y*z*y + z, y^2 - y*x*y + 1, z^2 - y*x*y - x*z*x>;
> A;
Finitely Presented Algebra of rank 3 over Rational Field
Non-commutative Graded Lexicographical Order
Variables: x, y, z
Quotient relations:
[
    y^4 - 7/2*z^2*x + z^3 - 1/2*x^2 + 2*y^2 + z*x + z^2 + x + 1,
    z*y*x^2 + y^3 - z^2*y + y,
    z^2*y*x - 1/2*z*y*z - z^2*y - 1/2*y*x,
    z^2*y*z - 3/2*y*x^2 - 3/4*z*y*z - 3/2*z^2*y - 3/4*y*x - y*z - z*y,
    z^3*x - 3/2*z^3 - 1/2*z*x,
    z^3*y - 3/2*y*x^2 - 3/4*z*y*z - 3/2*z^2*y - 3/4*y*x - y*z - z*y,
    z^4 - 2*z^2*x - 9/4*z^3 + 3/2*y^2 - 3/4*z*x - 1/2*z^2 + x + 3/2,
    x^3 + 3/2*z^2*x - z^3 + 1/2*x^2 + z*x,
    y^2*x - y^2 - 1,
    y^2*z + 3/2*z^2*x - z^3 + 1/2*x^2 + z,
    y*z*x - z*y*x,
    y*z*y - x^2 - z,
    y*z^2 - z^2*y,
    z*x^2 + y^2 - z^2 + 1,
    z*y^2 + 3/2*z^2*x - z^3 + 1/2*x^2 + z,
    x*y - y*x,
    x*z - z*x
]
> IsCommutative(A);
false
> y*z;
y*z
> z*y;
z*y
> U<u> := PolynomialRing(K);
> MinimalPolynomial(x);
u^8 - 7/2*u^7 + 4*u^6 - 3/2*u^5 - 1/2*u^4 + 2*u^3 - 2*u^2
> MinimalPolynomial(y);
u^16 - u^12 + 3*u^10 + u^8 - 10*u^6 - 15*u^4 - 9*u^2 - 2
> Dimension(A);
18
> V, Vf := VectorSpace(A);
> V;
Full Vector space of degree 18 over Rational Field
> Vf(x);
(0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0)
> [V.i@@Vf: i in [1 .. Dimension(V)]];
[
    1,
    z,
    y,
    x,
    z^2,
    z*y,
    z*x,
    y*z,
    y^2,
    y*x,
    x^2,
    z^3,
    z^2*y,
    z^2*x,
    z*y*z,
    z*y*x,
    y^3,
    y*x^2
]
> M, Mf := MatrixAlgebra(A);
> M;
Matrix Algebra of degree 18 with 4 generators over Rational Field
> M.1;
[0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0]
```

```
[0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0]
[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0]
[-1 0 0 0 1 0 0 0 -1 0 0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0]
[1 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1]
[0 0 0 0 0 0 -1 0 0 0 -1/2 1 0 -3/2 0 0 0 0]
[0 0 0 0 0 0 1/2 0 0 0 0 3/2 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0 0 1/2 0 0 1 0 1/2 0 0 0]
[0 0 0 0 0 0 0 0 0 0 1/2 0 0 3/2 0 0 0 0]
[0 0 0 0 0 0 0 0 0 1/2 0 0 1 0 1/2 0 0 0]
[0 0 -1 0 0 0 0 0 0 0 0 0 1 0 0 0 -1 0]
[0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0]
[0 0 0 0 0 1 0 1 0 0 0 0 0 0 0 -1 0 1]
> M.1 eq RepresentationMatrix(x);
true
> MinimalPolynomial(M.1);
u^8 - 7/2*u^7 + 4*u^6 - 3/2*u^5 - 1/2*u^4 + 2*u^3 - 2*u^2

> FactoredMinimalPolynomial(M.1);
[
    <u, 2>,
    <u^6 - 7/2*u^5 + 4*u^4 - 3/2*u^3 - 1/2*u^2 + 2*u - 2, 1>
]
> N := Kernel(M.1);
> N;
Vector space of degree 18, dimension 4 over Rational Field
Echelonized basis:
(0 1 0 0 0 0 0 0 0 0 3/4 -1/2 0 3/4 0 0 0 0)
(0 0 0 1 3 0 0 0 0 0 0 0 0 -2 0 0 0 0)
(0 0 0 0 0 1 0 -1 0 0 0 0 0 0 0 0 0 0)
(0 0 0 0 0 0 0 0 0 0 0 0 1 0 -1 0 0 0)
> a := N.1 @@ Vf;
> a;
3/4*z^2*x - 1/2*z^3 + 3/4*x^2 + z
> IsNilpotent(a);
true 3
> a^3;
0
> S, Sf := Algebra(A);
> S;
Associative Algebra of dimension 18 with base ring Rational Field
> Centre(S);
Associative Algebra of dimension 15 with base ring Rational Field
> J := JacobsonRadical(S);
> J;
Associative Algebra of dimension 4 with base ring Rational Field
```

```
> L := [J.i@@Sf: i in [1 .. 4]];
> L;
[
    3/4*z^2*x - 1/2*z^3 + 3/4*x^2 + z,
    -2*z^2*x + 3*z^2 + x,
    -y*z + z*y,
    -z*y*z + z^2*y
]
> [MinimalPolynomial(x): x in L];
[
    u^3,
    u^2,
    u^3,
    u^2
]
```

82.14 Vector Enumeration

Vector enumeration (originally misnamed module enumeration) is an algorithm for converting a finitely-presented module for a finitely-presented algebra into a concrete vector space on which the algebra has explicit matrix action. The algebra may be the group algebra of an fp-group, in which case the resulting module will be a matrix representation of the group, or it might be a more general fp-algebra, such as a Hecke algebra or a quotient of a polynomial ring.

82.14.1 Finitely Presented Modules

For a ring R, let M be an R-module M, generated as an R-module by s elements $\{m_1, \ldots, m_s\}$. There is an R-module epimorphism $\psi : R^s \mapsto M$, given by

$$(r_1, \ldots, r_s) \mapsto m_1 r_1 + \cdots + m_s r_s.$$

This shows that M is isomorphic to $A^s / \ker \psi$.

If $\ker \psi$ is generated as an R-module by a finite set L then we will say that M is presented by s generators and the relators L.

82.14.2 S-algebras

Suppose there is another ring S, equipped with a ring homomorphism $\phi : S \mapsto R$, such that $\phi(S)$ is central in R. In this situation any R-module can be described as an S-module, on which R acts as a ring of S-module endomorphisms. We say that R is an S-algebra. In particular, when S is a field k, any R-module is a k-vector space. If such a module V has finite k-dimension n, then V is characterised by this dimension and R will act on it as a subring of $M_n(k)$.

82.14.3 Finitely Presented Algebras

Given a finite set X, and a ring S, we can define the free S-algebra A generated by X. This can be seen either as the monoid algebra of the free monoid of words in X, or as all expressions in X and k, combined by addition and multiplication.

Given a finite set $R \subset A$ we can define

$$P = \frac{A}{\langle ARA \rangle}.$$

We say that P is a *finitely-presented* S-algebra, with generators X and relators R, and write it

$$P = \langle X \mid R \rangle .$$

The monoid algebra of any finitely-presented monoid (or group, of course) is finitely-presented, since

$$k \langle X \mid l_1 = r_1, \ldots, l_k = r_k \rangle_{\text{monoid}} = \langle X \mid l_1 - r_1, \ldots, l_k - r_k \rangle_{k-\text{algebra}} .$$

Furthermore, any quotient of a finitely-presented algebra by a finitely-generated two-sided ideal is finitely-presented. This gives us the general form of a finitely-presented algebra in Magma, as the quotient of the monoid algebra of an fp-monoid, by the two-sided ideal generated by some additional relators.

82.14.4 Vector Enumeration

The vector enumeration algorithm explicitly reconciles these two descriptions of an R-module, in the case where R is a finitely presented k-algebra for a field k, and M is a finitely presented R-module, which also has finite k-dimension.

Given the presentation of R as k-algebra, and that of M as R-module, it computes the k-dimension of M and the matrices giving the action of the generators of R on M. If M has infinite k-dimension the algorithm will fail to terminate.

Example H82E8

We consider some examples in the abstract. Below we will see how these and other calculations may be performed in Magma.

(1) **A permutation module**

In practice, it is always better to use the classical Todd-Coxeter algorithm to construct permutation representations of groups.

We know that the group with presentation

$$\left\langle a, b \mid a^4 = b^2 = (ab)^2 = 1 \right\rangle$$

is the dihedral group of order 8. Its group algebra over any field k is thus the fp-k-algebra

$$\left\langle a, b, a', b' \mid aa' - 1, a'a - 1, bb' - 1, b'b - 1, a^4 - 1, b^2 - 1, (ab)^2 - 1 \right\rangle .$$

The permutation module of degree 4 of this algebra is presented by one generator (as it is transitive) and the submodule generator $b - 1$.
Applying the algorithm to this case we obtain the matrices

$$a = \begin{pmatrix} 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 \end{pmatrix} \qquad b = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

and their inverses for a' and b'.
(2) **A quotient of a permutation module**
Like all permutation modules, this one fixes the all-ones vector $(1, 1, 1, 1)$, which we can see to be an image of $1 + a'(1 + b + a')$. We can construct the quotient of the permutation module by this 1-dimensional submodule by adding that word to the submodule generators. This gives

$$a = \begin{pmatrix} 0 & 0 & 1 \\ 1 & 0 & 0 \\ -1 & -1 & -1 \end{pmatrix} \qquad b = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \end{pmatrix}.$$

(3) **A non-cyclic module**
A permutation module of a group-ring is cyclic (that is, generated as a module by one element) just when the permutation representation is transitive. An intransitive permutation representation can be easily constructed from its transitive components, but in general it is not so easy to construct an arbitrary module from cyclic submodules. Accordingly it can be worthwhile to construct non-cyclic modules directly.
As an example we take two copies of the representation constructed in example one, and fuse their one-dimensional submodules. The generators and relators are as before, and now $s = 2$ and the submodule generators are $a^2 = \{(b - 1, 0), (0, b - 1), (1 + a'(1 + a' + b), -1 - a'(1 + a' + b))\}$. We obtain a representation of degree seven, given by

$$a = \begin{pmatrix} 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & -1 & 1 & 1 & 1 & -1 & -1 \end{pmatrix} \qquad b = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{pmatrix}.$$

82.14.5 The Isomorphism

In determining the k-dimension of M, and giving matrices representing the action of R on M, the algorithm is, in effect, constructing a k-vector space, on which R has matrix action, and which is R-module isomorphic to M, which is formally a quotient module of R^s. The algorithm also computes this isomorphism, giving images in the vector space for the s standard generators of R^s and pre-images in R^s of the basis of the vector space.

In example (1) above, we find that the module generator has image $(1, 0, 0, 0)$, while the basis vectors have pre-images 1, a', a'^2 and b.

In example (3) the images of the two module generators are $(1, 0, 0, 0, 0, 0, 0)$ and $(0, 1, 0, 0, 0, 0, 0)$ while the basis vectors are images of $(1, 0)$, $(0, 1)$, $(a', 0)$,$(a, 0)$, $(a^2, 0)$, $(0, a')$ and $(0, a)$.

82.14.6 Sketch of the Algorithm

The algorithm is based on the Haselgrove-Leech-Trotter (HLT) version of the Todd-Coxeter algorithm, which we consider as a means of constructing the permutation representation of a finitely-presented group on the cosets of a finitely generated subgroup.

The algorithm proceeds by manipulating a partial action of the free algebra on a vector space. This is represented as a table, with columns indexed by the generators of the algebra, and rows indexed by the basis of the space. Each entry is either a vector or $\perp$, signifying that no action is given.

We can extend this to a "complete action with side-effects", by adding a new row to the table whenever needed. We call this *the action with defining*.

The action of the fp-algebra P on M gives an action for the free algebra A, and our objective is to modify out partial action until it becomes this action. We know certain things about this action, which drive our modification process:

1. It is a complete action.
2. The relators of P annihilate every vector.
3. The images in the space of the relators of M are zero.
4. The space contains images of the R-module generators of M.

The algorithm begins by applying the fourth fact and creating s linearly independent vectors. It then applies the third, computing the action with defining of the relators of M (the set called L above). The fact that these images should be zero gives a linear dependence among our basis vectors (called a coincidence), which we use to reduce their number. As we do this, we have to take care not to lose the information already in the table, which may give rise to further coincidences.

We now start to exploit the second fact about M, constructing the action (with defining) of the relators of P, on the basis vectors, and applying the resulting coincidences. This process may not terminate, as we are defining new basis vectors on the one hand, and removing them through coincidences on the other. It is possible to prove, however, that if M is in fact finite-dimensional then the process will terminate.

The first fact is applied by adding the relators $x - x$ for each $x \in X$ to the relators of P, so that the image of every basis vector under each x is sure to be defined.

82.14.7 Weights

The above sketch leaves open the question of the order in which the relators are applied to the basis vectors. The proof that the algorithm completes when M is finite-dimensional imposes some rather loose constraints, but within these constraints there is considerable choice.

The implementation in MAGMA uses a system of weights. Each relator r is assigned a weight w_r by the user, and each basis vector e has a weight w_e. There is a current weight w, which increases as the computation progresses, and at weight w all basis vector, relator pairs (b, r) such that $w_b + w_r \leq w$, which have not been processed already, are processed. New basis vectors defined during this process are assigned weight w. The initial basis vectors and those defined during processing of the submodule generators are assigned weight 1. See below for details of how to set the weights, and their default values.

82.14.8 Setup Functions

For V2.11, the following functions create the old representation for fp-algebras which are necessary for the Vector Enumeration algorithm. These are compatible with older versions of Magma. See the examples below for examples of how to use these functions.

```
FreeAlgebra(R, M)
```

```
FreeAlgebra(R, G)
```

Construct the special fp-algebra over the ring R and the monoid M or the group G, for use in the Vector Enumeration algorithm.

82.14.9 The Quotient Module Function

```
QuotientModule(A, S)
```

Given an fp-k-algebra A, for a field k, with r generators, and a submodule N of the free A-module of rank s specified by S, construct an A-module isomorphic to the quotient module A^s/N together with the isomorphism.

The three values returned are:

M a sequence of r $n \times n$ matrices with entries in k;

I a sequence of s vectors of length n with entries in k;

P a sequence of n elements of the free A-module A^s.

The matrices M specify a homomorphism from A to $\mathrm{End}_k(k^n)$, under which k^n becomes an A-module isomorphic to A^s/N. That isomorphism is given in one direction by the vectors I, which are the images in k^n of the s generators $(1, 0, \ldots, 0)$, $(0, 1, 0, \ldots, 0)$, $\ldots$, $(0, \ldots, 0, 1)$ of A^s. In the other direction, the elements P give representatives for the images in A^s/N of the images of the n standard basis elements of k^n.

The submodule N may be specified by the parameter S in three ways:

(1) S may simply be N, a finitely generated submodule of a free A-module (except that such an object cannot currently be created).

(2) S may be a finitely generated right ideal of A, in which case $s = 1$ and N is S considered as a submodule of A considered as a right module for itself.

(3) S may be a sequence of elements of a free A-module A^s, in which case N is the submodule that they generate (this is a stand in for 1 and could be removed when 1 is implemented).

82.14.10 Structuring Presentations

The relations used by the vector enumeration algorithm come from three sources:

(1) The relations of the fp-group or fp-monoid underlying A.

(2) The relations of A itself.

(3) The generators of N.

The third group play the same role as subgroup generators in the Todd-Coxeter algorithm, and are treated specially, while the first two groups are logically equivalent, forming the relations of A in the variety of free finitely-generated associative algebras. However, when the underlying monoid of A is actually an fp-group G, the vector enumeration algorithm can use a more efficient technique to process the relations of G, and can take advantage of the fact that the generators of G are known to be invertible.

This greatly improves the performance of the algorithm, and so users are recommended to ensure as far as possible that:

(1) If the underlying monoid of an fp-algebra is in fact an fp-group then it should be presented to Magma as such.

(2) Relations which can be written as equations between monomials should be given as relations of the underlying monoid, rather than as relations of the algebra.

82.14.11 Options and Controls

The `QuotientModule` function supports a large selection of optional arguments.

82.14.12 Weights

The processing of the relations of A by the vector enumeration algorithm depends on weights which are assigned to those relations (see above). The higher the weight of a relation the later it will be processed. By default, all relations are given weight 3, except those arising from relators of an fp-group, which are given weight equal to half the length of the relator.

Separate weights are used in lookahead mode, with the same default values.

`QuotientModule(A, S)`

`MonomialWeights` [RngIntElt] *Default :*

`MonWts` [RngIntElt] *Default :*

This option sets the sequence of weights for the relations derived from the relations of the underlying monoid of A. The weights w_1, w_2, etc. are applied to the relations in the order in which they appear. If there are fewer weights than relations the remaining relations are assigned the default weight; if there are more weights than relations the extra weights are silently discarded.

Unless the `MonomialLookaheadweights` or `MonLWts` parameters are present, these weights are also used in lookahead mode.

`MonomialLookaheadWeights`

[RngIntElt] *Default :*

`MonLWts` [RngIntElt] *Default :*

This option sets the sequence of weights for the relations derived from the relations of the underlying monoid of A in lookahead mode only. It is otherwise similar to `MonomialWeights`.

`AlgebraWeights` [RngIntElt] *Default :*

`AlgWts` [RngIntElt] *Default :*

This option sets the sequence of weights for the relations given explicitly as relations of the algebra A. It is otherwise similar to `MonomialWeights`.

`AlgebraLookaheadWeights`

[RngIntElt] *Default :*

`AlgLWts` [RngIntElt] *Default :*

This option sets the sequence of weights for the relations given explicitly as relations of the algebra A in lookahead mode only. It is otherwise similar to `AlgebraWeights`.

82.14.13 Limits

`QuotientModule(A, S)`

Options in this group set limits on the progress of the algorithm. If the calculation cannot be performed under these constraints the value `undef` is returned, unless the `ErrorOnFail` option was set, in which case a run-time error is generated.

`MaximumDimension` RngIntElt *Default :* ∞

`MaxDim` [RngIntElt] *Default :* ∞

This sets a limit of n on the dimension of the vector-space constructed, and on the dimension of the intermediate spaces used in the construction.

By default there is no limit, except for available memory.

`MaximumTime` FldReElt *Default :* ∞

`MaxTime` FldReElt *Default :* ∞

This sets a limit on the CPU time available for the vector enumeration. The limit is given as a real number t and is measured in seconds.

This limit is only checked at certain points in the calculation, so it is possible for a vector enumeration to over-run, possibly by a significant amount.

By default, there is no limit.

`MaximumWeight` RngIntElt *Default :* 100

`MaxWt` RngIntElt *Default :* 100

This sets a limit on the maximum weight of (basis vector, relation) pairs that will be used by the algorithm.

The weight of a basis vector is the weight of the pair that was being processed when it was defined. The weight of a pair is the weight of the basis vector plus the weight of the relation (see above).

The default limit is 100.

82.14.14 Logging

`QuotientModule(A, S)`

There are a number of options to control the level of detail provided in the informational message from the vector enumerator. When multiple contradictory options are given the first one given takes precedence.

`NoLogging`	BoolElt	*Default :* `false`
`NoLog`	BoolElt	*Default :* `false`
`Silent`	BoolElt	*Default :* `false`

This option turns off all the informational messages produced by the vector enumerator.

`MaximumLogging`	BoolElt	*Default :* `false`
`MaxLog`	BoolElt	*Default :* `false`

This option turns on the highest possible level of detail in the informational messages. This is *too detailed* for almost all purposes except debugging.

`LogActions`	RngIntElt	*Default :* 0
`LogAct`	RngIntElt	*Default :* 0

This option sets the level of messages about the computation of the action of the algebra on the vector space under construction. At level 0 (the default) no messages are produced. All other levels produce copious output, with all levels above 2 being equivalent.

`LogCoincidences`	RngIntElt	*Default :* 0
`LogCoin`	RngIntElt	*Default :* 0

This option sets the level of messages about the coincidences discovered and the processing of them. At level 0 (the default) no messages are produced. At level 1 every coincidence and deduction is recorded when it is discovered and when it is processed. At level 2 or higher the operation of finding the undeleted image of a vector is also recorded.

`LogInitialization`	RngIntElt	*Default :* 0
`LogInitialisation`	RngIntElt	*Default :* 0
`LogInit`	RngIntElt	*Default :* 0

This option sets the level of messages about the initialisation of new basis vectors. At level 0 (the default) no messages are produced. All other levels produce a message whenever a new basis vector is defined.

`LogPacking`	RngIntElt	*Default :* 1
`LogPack`	RngIntElt	*Default :* 1

This option sets the level of messages about the reclamation of free space in the tables used by the algorithm. At level 0 no messages are produced. At level 1 (the default) it produces a message each time the pack routine is called. At level 2 or higher it records the exact renaming used to reclaim the space.

`LogPushes`	RngIntElt	*Default :* 0
`LogPush`	RngIntElt	*Default :* 0

This option sets the level of messages about the pushing (or tracing) of (basis vector, relation) pairs. At level 0 (the default) it produces no messages. At level 1 a message is produced for each push that is started. At level 2 or higher the outcome of the push is also recorded.

`LogProgress`	RngIntElt	*Default :* 0
`LogStages`	RngIntElt	*Default :* 0

This option sets the level of messages about the overall progress of the algorithm. At level 0 no messages are produced. At level 1, simple messages are printed as the algorithm passes through its major stages. At level 2 the relations are printed as they are read in, and the complete action on the final module is printed. At level 3 or higher the action is also printed after the processing of submodule generators is complete.

`LogWeightChanges`	RngIntElt	*Default :* 1
`LogWt`	RngIntElt	*Default :* 1

This option sets the level of messages about changes in the current weight (ie the weight of (basis vector, relation pairs) currently being pushed. At level 0 no such messages are produced. At level 1 (the default) or higher a message giving the new weight and the current dimension is printed.

82.14.15 Miscellaneous

`QuotientModule(A, S)`

`Lookahead`	BoolElt	*Default :* `true`

This option controls whether, and to what extent, lookahead is used. If x is `false` then lookahead is not used. If x is `true`, the default, the lookahead by the default amount (two weights) is used. If x is a positive integer n then lookahead n weights is used. A sufficiently large value of n is equivalent to complete lookahead. Lookahead is commenced approximately every time the dimension doubles.

`EarlyClosing`	BoolElt	*Default :* `false`
`Early`	BoolElt	*Default :* `false`

This option permits the algorithm to stop as soon as the table represents a complete action (but see below), without checking to see whether the action satisfies all the relations. In practice this action is usually correct. The default behaviour is to continue and check all the relations.

`EarlyClosingMinimum`	RngIntElt	*Default :*
`ECMin`	RngIntElt	*Default :*

This option sets a minimum dimension at which the algorithm may stop without checking all the relators. It implies `EarlyClosing`.

`EarlyClosingMaximum`	RngIntElt	*Default :*
`ECMax`	RngIntElt	*Default :*

This option sets a maximum dimension at which the algorithm aamy stop without checking all the relators. It implies `EarlyClosing`.

`ConstructMorphism`	BoolElt	*Default :* `true`
`Morphism`	BoolElt	*Default :* `true`

This option controls whether the third return value of the `QuotientModule` function is in fact computed. A small overhead of time and space is required to compute it, and many applications do not need it, so this option is provided. When *b* is `true` (the default) the third return value is computed, when *b* is `false` it is not.

`ErrorOnFail`	BoolElt	*Default :*
`ErrFail`	BoolElt	*Default :*

This option controls the behaviour of the program if there is insufficient time or space to complete the calculation, or if the calculation has not been completed when the maximum weight is reached. If it is present a run-time error is generated, otherwise the value `undef` is returned.

Example H82E9

First we repeat the examples above, in Magma. The permutation action of D_8

```
> d8<a,b> := Group<a,b | a^4 = b^2 = (a*b)^2 = 1>;
> q := RationalField();
> a1<a,b> := FreeAlgebra(q,d8);
> i1 := rideal<a1 | b-1 >;
> mats, im, preim := QuotientModule(a1,i1);
Read Input
Done submodule generators
Starting weight 2 in define mode, 1 alive out of 2
Starting weight 3 in define mode, 1 alive out of 2
Looking ahead ...
Starting weight 4 in lookahead mode, 4 alive out of 5
Starting weight 5 in lookahead mode, 4 alive out of 5
    ...done
Packing 5 to 4
Starting weight 4 in define mode, 4 alive out of 4
Starting weight 5 in define mode, 4 alive out of 4
Starting weight 6 in define mode, 4 alive out of 4
Starting weight 7 in define mode, 5 alive out of 6
Starting weight 8 in define mode, 6 alive out of 7
Starting weight 9 in define mode, 4 alive out of 7
Starting weight 10 in define mode, 4 alive out of 7
Closed, 7 rows defined
```

```
Packing 7 to 4
4 live dimensions
Successful
> mats;
[
    [0 0 1 0]
    [1 0 0 0]
    [0 0 0 1]
    [0 1 0 0],

    [1 0 0 0]
    [0 0 1 0]
    [0 1 0 0]
    [0 0 0 1]
]
> im;
[
    (1 0 0 0)
]
> preim;
[ Id(), a^-1, a^-1 * b, a^-2 ]
>
```

Example H82E10

A quotient of that module.
We continue from the last example and set:

```
> d8<a,b> := Group<a,b | a^4 = b^2 = (a*b)^2 = 1>;
> q := RationalField();
> a1<a,b> := FreeAlgebra(q,d8);
> i2 := rideal<a1 | b-1, 1+a^3+a^3*b+a^2>;
> mats, im, preim := QuotientModule(a1,i2);
Read Input
Done submodule generators
Starting weight 2 in define mode, 4 alive out of 6
Starting weight 3 in define mode, 5 alive out of 7
Starting weight 4 in define mode, 3 alive out of 7
Starting weight 5 in define mode, 3 alive out of 7
Closed, 7 rows defined
Packing 7 to 3
3 live dimensions
Successful
> mats;
[
    [ 0  1  0]
    [ 0  0  1]
    [-1 -1 -1],
    [ 1  0  0]
    [-1 -1 -1]
    [ 0  0  1]
]
```

82.15 Bibliography

[**CLO96**] David Cox, John Little, and Donal O'Shea. *Ideals, Varieties and Algorithms.* Undergraduate Texts in Mathematics. Springer, New York–Berlin–Heidelberg, 2nd edition, 1996.

[**Fau99**] Jean-Charles Faugère. A new efficient algorithm for computing Gröbner bases (F_4). *Journal of Pure and Applied Algebra*, 139 (1-3):61–88, 1999.

[**Li02**] Huishi Li. *Noncommutative Gröbner Bases and Filtered-Graded Transfer*, volume 1795 of *Lecture Notes in Math.* Springer-Verlag, Berlin–Heidelberg–New York, 2002.

[**Mor94**] Teo Mora. An introduction to commutative and noncommutative Gröbner bases. *Theoretical Computer Science*, 134:134–173, 1994.

83 MATRIX ALGEBRAS

Chapter 83
MATRIX ALGEBRAS

83.1 Introduction

Matrix algebras (or matrix rings) may be defined over any ring S. We shall regard such a matrix algebra as an S-*algebra*. Let us denote the complete algebra of $n \times n$ matrices over S by $M_n(S)$. It will often be convenient to regard $M_n(S)$ as the endomorphism ring of the free S-module $S^{(n)}$. We will then speak of $S^{(n)}$ as the natural S-module associated with $M_n(S)$.

Matrix algebras have type `AlgMat` and their elements have type `AlgMatElt`. These types inherit from `AlgMatV` and `AlgMatVElt` respectively (which cover both ordinary matrix algebras and matrix Lie algebras).

83.2 Construction of Matrix Algebras and their Elements

83.2.1 Construction of the Complete Matrix Algebra

```
MatrixAlgebra(S, n)
```

```
MatrixRing(S, n)
```

Given a positive integer n and a ring S, create the complete matrix algebra $M_n(S)$, consisting of all $n \times n$ matrices with coefficients in the ring S.

83.2.2 Construction of a Matrix

```
elt< R | L >
```

Given a matrix algebra defined as a subalgebra of $M_n(S)$, create the element of R defined by the list L of n^2 elements from S.

```
R ! Q
```

Given a matrix algebra R defined as a subalgebra of $M_n(S)$ and a sequence $Q = [a_{11}, \ldots, a_{1n}, a_{21}, \ldots, a_{2n}, \ldots, a_{n1}, \ldots, a_{nn}]$ of n^2 elements of S, return the matrix

$$\begin{pmatrix} a_{11} & a_{12} & \ldots & a_{1n} \\ a_{21} & a_{22} & \ldots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \ldots & a_{nn} \end{pmatrix}$$

as an element of R. Note that the algebra R must exist before an attempt is made to create matrices.

```
CambridgeMatrix(t, K, n, Q)
```

This function creates a n by n matrix over the finite field K of cardinality q specified in a "Cambridge" format in the general matrix algebra of degree n over K. The parameter t specifies the type of the format. If t is 1, then q is assumed to be less than 10 and the sequence Q must consist of n strings which give the n rows—each string must have length n and contain the entries of that row (each entry is a digit in the range $[0,\ q-1]$). If t is 3 then Q must consist of n^2 integers in the range $[0,\ q-1]$ which give the entries in row-major order. In either format, if $q = p^e$, where p is prime and $e > 1$, then an entry x is written as a vector using the base-p representation of length e of x and the corresponding element in K is used (see the Finite Fields chapter for details). This function is principally provided for the reading in of large matrices.

```
CompanionMatrix(p)
```

Given a monic polynomial p of degree n over a ring R, create the companion matrix C for p as an element of $M_n(R)$. The minimal and characteristic polynomial of C is then p.

```
DiagonalMatrix(R, Q)
```

If R is a subalgebra of $M_n(S)$ and Q is a sequence of n elements of S, create the diagonal matrix $diag(Q[1], Q[2], \ldots, Q[n])$.

```
MatrixUnit(R, i, j)
```

Create the matrix unit $E(i, j)$ in the matrix algebra R, i.e. the matrix having the one of the coefficient ring of R in position (i, j) and zeros elsewhere.

```
Random(R)
```

Create a random matrix of the matrix algebra R.

```
ScalarMatrix(R, t)
```

If R is a subalgebra of $M_n(S)$ and t is an element of the ring S, create the scalar matrix $t * I$ in R.

```
R ! 1
```

Create the identity matrix I_n of the matrix algebra R.

```
R ! 0
```

Create the zero matrix of the matrix algebra R.

```
R ! t
```

Create the scalar matrix $t * I$ of the matrix algebra R.

83.2.3 Constructing a General Matrix Algebra

```
MatrixAlgebra< S, n | L >
```

```
MatrixRing< S, n | L >
```

Given a commutative ring S and a positive integer n, create the S-algebra R consisting of the $n \times n$ matrices over the ring S generated by the elements defined in the list L. Let F denote the algebra $M_n(S)$. Each term L_i of the list L must be an expression defining an object of one of the following types:

(a) A sequence of n^2 elements of S defining an element of F.

(b) A set or sequence whose terms are sequences of type (a).

(c) An element of F.

(d) A set or sequence whose terms are elements of F.

(e) The null list.

The generators stored for R consist of the elements specified by terms L_i together with the stored generators for subalgebras specified by terms of L_i. Repetitions of an element and occurrences of scalar matrices are removed.

Example H83E1

We demonstrate the use of the matrix algebra constructor by creating an algebra of 3 × 3 lower-triangular matrices over the rational field.

```
> Q := RationalField();
> A := MatrixAlgebra< Q, 3 | [ 1/3,0,0, 3/2,3,0, -1/2,4,3],
>        [ 3,0,0, 1/2,-5,0, 8,-1/2,4] >;
> A:Maximal;
Matrix Algebra of degree 3 with 2 generators over Rational Field
Generators:
[ 1/3    0    0]
[ 3/2    3    0]
[-1/2    4    3]

[   3    0    0]
[ 1/2   -5    0]
[   8 -1/2    4]
> Dimension(A);
6
```

Example H83E2

We construct a 4 by 4 matrix over the finite field with 5 elements using the CambridgeMatrix function.

```
> K := FiniteField(5);
> x := CambridgeMatrix(1, K, 4, [ "1234", "0111", "4321", "1211" ]);
> x;
[1 2 3 4]
[0 1 1 1]
[4 3 2 1]
[1 2 1 1]
```

83.2.4 The Invariants of a Matrix Algebra

`R . i`

The i-th defining generator for the matrix algebra R.

`BaseRing(R)`

`CoefficientRing(R)`

The coefficient ring S for the matrix algebra R.

`Degree(R)`

Given a matrix algebra R, return the degree n of R.

`Generators(R)`

The set consisting of the defining generators for the matrix algebra R.

`Generic(R)`

The complete matrix algebra $M_n(S)$ in which the matrix algebra R is naturally embedded.

`BaseModule(R)`

If R is a subring of the matrix algebra $M_n(S)$, then R is considered to act on the free S-module of rank n, consisting of n-tuples over S. The function `BaseModule` returns this S-module.

`NumberOfGenerators(R)`

`Ngens(R)`

The number of defining generators for the matrix algebra R.

`Parent(a)`

Given an element a belonging to the matrix algebra R, return R, i.e. the parent structure for a.

Example H83E3

We illustrate the use of these functions by applying them to the algebra of 3 × 3 lower-triangular matrices over the rational field constructed above.

```
> Q := RationalField();
> A := MatrixAlgebra< Q, 3 | [ 1/3,0,0, 3/2,3,0, -1/2,4,3],
>        [ 3,0,0, 1/2,-5,0, 8,-1/2,4] >;
> CoefficientRing(A);
Rational Field
> Degree(A);
3
> Ngens(A);
2
> Generators(A);
{
    [ 1/3    0    0]
    [ 3/2    3    0]
    [-1/2    4    3],

    [   3    0    0]
    [ 1/2   -5    0]
    [   8 -1/2    4]
}
> Generic(A);
Full Matrix Algebra of degree 3 over Rational Field
> Dimension(A);
6
```

83.3 Construction of Subalgebras, Ideals and Quotient Rings

```
sub< R | L >
```

Given the matrix algebra R, defined as a subring of $M_n(S)$, construct the subring T of R generated by the elements specified by the list L, where L is a list of one or more items of the following types:

(a) A sequence of n^2 elements of S defining an element of R;

(b) An element of R;

(c) A set or sequence of elements of R;

(d) A subring of R;

(e) A set or sequence of subrings of R.

Each element or subalgebra specified by the list must belong to the *same* complete matrix algebra. The subalgebra T will be constructed as a subalgebra of some matrix algebra which contains each of the elements and subalgebras specified in the list.

The generators of T consist of the elements specified by the terms of the list L together with the stored generators for subalgebras specified by terms of the list. Repetitions of an element and occurrences of the identity element are removed (unless T is trivial).

The constructor returns the subalgebra T and the inclusion homomorphism $f : T \to R$.

```
ideal< R | L >
```

Given the matrix algebra R, construct the two-sided ideal I of R generated by the elements of R specified by the list L, where the possibilities for L are the same as for the `sub`-constructor.

```
lideal< R | L >
```

Given the matrix algebra R, construct the left ideal I of R generated by the elements of R specified by the list L, where the possibilities for L are the same as for the `sub`-constructor.

```
rideal< R | L >
```

Given the matrix algebra R, construct the right ideal I of R generated by the elements of R specified in the list L, where the possibilities for L are the same as for the `sub`-constructor.

Example H83E4

We construct the subalgebra of the matrix algebra A (defined above) that is generated by the first generator.

```
> Q := RationalField();
> A := MatrixAlgebra< Q, 3 | [ 1/3,0,0, 3/2,3,0, -1/2,4,3],
>        [ 3,0,0, 1/2,-5,0, 8,-1/2,4] >;
> B := sub< A | A.1 >;
> Dimension(B);
3
> B: Maximal;
Matrix Algebra of degree 3 and dimension 3 with 1 generator
over Rational Field
Generators:
[ 1/3    0    0]
[ 3/2    3    0]
[-1/2    4    3]

Basis:

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

```
[0 0 1]

[    0     0     0]
[    1  16/9     0]
[    0 88/27  16/9]

[   0    0    0]
[   0    0    0]
[   1 16/9    0]
```

83.4 The Construction of Extensions and their Elements

83.4.1 The Construction of Direct Sums and Tensor Products

`DirectSum(R, T)`

Given two matrix algebras R and T, where R and T have the same coefficient ring S, return the direct sum D of R and T (with the action given by the direct sum of the action of R and the action of T).

`TensorProduct(A, B)`

Given two unital matrix algebras A and B, where A and B have the same coefficient ring S, construct the tensor product of A and B.

Example H83E5

We construct the direct product and tensor product of the matrix algebra A (defined above) with itself.

```
> Q := RationalField();
> A := MatrixAlgebra< Q, 3 | [ 1/3,0,0, 3/2,3,0, -1/2,4,3],
>         [ 3,0,0, 1/2,-5,0, 8,-1/2,4] >;
> AplusA := DirectSum(A, A);
> AplusA: Maximal;
Matrix Algebra of degree 6 with 4 generators over
    Rational Field
Generators:
[ 1/3    0    0    0    0    0]
[ 3/2    3    0    0    0    0]
[-1/2    4    3    0    0    0]
[   0    0    0    0    0    0]
[   0    0    0    0    0    0]
[   0    0    0    0    0    0]

[   3    0    0    0    0    0]
[ 1/2   -5    0    0    0    0]
[   8 -1/2    4    0    0    0]
[   0    0    0    0    0    0]
[   0    0    0    0    0    0]
[   0    0    0    0    0    0]

[   0    0    0    0    0    0]
[   0    0    0    0    0    0]
[   0    0    0    0    0    0]
[   0    0    0  1/3    0    0]
[   0    0    0  3/2    3    0]
[   0    0    0 -1/2    4    3]

[   0    0    0    0    0    0]
[   0    0    0    0    0    0]
[   0    0    0    0    0    0]
[   0    0    0    3    0    0]
[   0    0    0  1/2   -5    0]
[   0    0    0    8 -1/2    4]
> AtimesA := TensorProduct(A, A);
> AtimesA: Maximal;
Matrix Algebra of degree 9 with 4 generators over
    Rational Field
Generators:
[ 1/3    0    0    0    0    0    0    0    0]
[   0  1/3    0    0    0    0    0    0    0]
[   0    0  1/3    0    0    0    0    0    0]
[ 3/2    0    0    3    0    0    0    0    0]
[   0  3/2    0    0    3    0    0    0    0]
[   0    0  3/2    0    0    3    0    0    0]
[-1/2    0    0    4    0    0    3    0    0]
[   0 -1/2    0    0    4    0    0    3    0]
[   0    0 -1/2    0    0    4    0    0    3]

[   3    0    0    0    0    0    0    0    0]
[   0    3    0    0    0    0    0    0    0]
[   0    0    3    0    0    0    0    0    0]
[ 1/2    0    0   -5    0    0    0    0    0]
[   0  1/2    0    0   -5    0    0    0    0]
[   0    0  1/2    0    0   -5    0    0    0]
[   8    0    0 -1/2    0    0    4    0    0]
[   0    8    0    0 -1/2    0    0    4    0]
[   0    0    8    0    0 -1/2    0    0    4]

[ 1/3    0    0    0    0    0    0    0    0]
[ 3/2    3    0    0    0    0    0    0    0]
[-1/2    4    3    0    0    0    0    0    0]
[   0    0    0  1/3    0    0    0    0    0]
[   0    0    0  3/2    3    0    0    0    0]
[   0    0    0 -1/2    4    3    0    0    0]
[   0    0    0    0    0    0  1/3    0    0]
[   0    0    0    0    0    0  3/2    3    0]
[   0    0    0    0    0    0 -1/2    4    3]

[   3    0    0    0    0    0    0    0    0]
[ 1/2   -5    0    0    0    0    0    0    0]
[   8 -1/2    4    0    0    0    0    0    0]
[   0    0    0    3    0    0    0    0    0]
[   0    0    0  1/2   -5    0    0    0    0]
[   0    0    0    8 -1/2    4    0    0    0]
[   0    0    0    0    0    0    3    0    0]
[   0    0    0    0    0    0  1/2   -5    0]
[   0    0    0    0    0    0    8 -1/2    4]
```

83.4.2 Construction of Direct Sums and Tensor Products of Elements

`DirectSum(a, b)`

Given an element a of the matrix algebra Q and an element b of the matrix algebra R, form the direct sum of matrices a and b. The square is returned as an element of the matrix algebra T, which must be the direct sum of the parent of a and the parent of b.

`ExteriorSquare(a)`

Given an element a of the matrix algebra $M_n(S)$, form the exterior square of a as an element of $M_m(S)$, where $m = n(n-1)/2$.

`ExteriorPower(a,r)`

Given an element a of the matrix algebra $M_n(S)$, form the rth exterior power of a as an element of $M_m(S)$, where $\binom{m=n}{r}$.

`SymmetricSquare(a)`

Given an element a of the matrix algebra $M_n(S)$, form the symmetric square of a as an element of $M_m(S)$, where $m = n(n+1)/2$.

`SymmetricPower(a,r)`

Given an element a of the matrix algebra $M_n(S)$, form the rth symmetric power of a as an element of $M_m(S)$, for the appropriate m.

`TensorProduct(a, b)`

Given an element a belonging to a subalgebra of $M_{n_1}(S)$ and an element b belonging to a subalgebra of $M_{n_2}(S)$, construct the tensor product of a and b as an element of the matrix algebra $M_n(S)$, where $n = n_1 * n_2$.

83.5 Operations on Matrix Algebras

`Centre(A)`

Given a matrix algebra A whose base ring is a field, return the centre of A.

`Centralizer(A, S)`

Given a matrix algebra A whose base ring is a field, together with a subalgebra S of A return the centralizer of S in A.

83.6 Changing Rings

`ChangeRing(A, S)`

Given a matrix algebra A with base ring R, together with a ring S, construct the matrix algebra B with base ring S obtained by coercing the components of elements of A into S, together with the homomorphism from A to B.

`ChangeRing(A, S, f)`

Given a matrix algebra A with base ring R, together with a ring S and a homomorphism $f : R- > \ S$, construct the matrix algebra B with base ring S obtained by mapping the components of elements of R into S by f, together with the homomorphism from A to B.

`hom<  A -> B | f  >`

Given *full* matrix algebras A and B, together with a homomorphism f from the base ring R of A to the base ring S of B, create the homomorphism from A to B which, given a matrix in A, applies f to the entries of the matrix.

83.7 Elementary Operations on Elements

83.7.1 Arithmetic

`a + b`

Sum of the matrices a and b, where a and b belong to a common matrix algebra R.

`a + t`
`t + a`

Sum of the matrix a and the scalar matrix $t * I$.

`-a`

Negation of the matrix a.

`a - b`

Difference of the matrices a and b, where a and b belong to the same matrix algebra R.

`a - t`

`t - a`

Difference of the matrix a and the scalar matrix $t * I$.

`a * b`

Product of the matrices a and b, where a and b belong to the same matrix algebra R.

`a * b`

Given a matrix a belonging to a subalgebra of $M_n(S)$ and an element b of a submodule of $\mathrm{Hom}(R^{(n)}, R^{(m)})$, construct the product of a and b as an element of $\mathrm{Hom}(R^{(n)}, R^{(m)})$.

`a * b`

Given a matrix a belonging to a submodule of $\mathrm{Hom}(R^{(n)}, R^{(m)})$ and an element b of a subalgebra of $M_m(S)$, construct the product of a and b as an element of $\mathrm{Hom}(R^{(n)}, R^{(m)})$.

`t * a`

`a * t`

Given an element a of the matrix algebra R, and an element t belonging to the coefficient ring S of R, form their scalar product.

`u * a`

Given an element u belonging to the S-module $S^{(n)}$ and an element a belonging to a subalgebra of $M_n(S)$, form the element $u * a$ of S^n.

`a ^ n`

If n is positive, form the n-th power of a; if n is zero, form the identity matrix; if n is negative, form the $(-n)$-th power of the inverse of a.

`NumberOfColumns(a)`

`Ncols(a)`

The number of columns in the matrix a.

`NumberOfRows(a)`

`Nrows(a)`

The number of rows in the matrix a.

83.7.2 Predicates

83.7.2.1 Comparison

```
a eq b
```

Returns `true` if the matrix a is equal to the matrix b, where a and b are elements of a common matrix algebra R.

```
a ne b
```

Returns `true` if the matrix a is not equal to the matrix b, where a and b are elements of a common matrix algebra R.

83.7.2.2 Properties of Elements

The functions given here test properties of matrices. See also the section in the Lattices chapter for a description of the function `IsPositiveDefinite` and related functions.

```
IsDiagonal(a)
```

Returns `true` iff the element a belonging to the matrix algebra R is a diagonal matrix; i.e. the only non-zero entries are on the diagonal.

```
IsMinusOne(a)
```

Returns `true` iff the element a belonging to the matrix algebra R is the negation of the identity element for R.

```
IsOne(a)
```

Returns `true` iff the element a belonging to the matrix algebra R is the identity element for R.

```
IsScalar(a)
```

Returns `true` iff the element a belonging to the matrix algebra R is a scalar matrix.

```
IsSymmetric(a)
```

Returns `true` iff the element a belonging to the matrix algebra R is a symmetric matrix; i.e. the transpose of a equals a.

```
IsUnit(a)
```

Returns `true` iff the matrix a belonging to the matrix algebra R is a unit.

```
IsZero(a)
```

Returns `true` iff the element a belonging to the matrix algebra R is the zero element for R.

```
IsNilpotent(a)
```

Return `true` if some power of the matrix a belonging to a matrix algebra is the zero of the matrix algebra. Also returns the minimum exponent n such that $a^n = 0$.

```
IsUnipotent(a)
```

Return `true` if the matrix a belonging to a matrix algebra is the identity of that algebra plus a nilpotent matrix. Also returns the index of nilpotence of $a - I$.

```
Rank(a)
```

Return the rank of the element a belonging to the matrix algebra R.

```
Determinant(A)
```

`MonteCarloLevel`	RngIntElt	*Default :* 0
`Proof`	BoolElt	*Default :* `true`
`pAdic`	BoolElt	*Default :* `true`
`Divisor`	RngIntElt	*Default :* 0

Given a square matrix A over the ring R, return the determinant of A as an element of R. R may be any commutative ring. The determinant of the 0×0 matrix over R is defined to be `R!1`.

If the coefficient ring is the integer ring $\mathbf{Z}$ or the rational field $\mathbf{Q}$ then a modular algorithm based on that of Abbott et al. [ABM99] is used, which first computes a divisor d of the determinant D using a fast p-adic nullspace computation, and then computes the quotient D/d by computing the determinant D modulo enough small primes to cover the Hadamard bound divided by d. This always yields a correct answer.

If the parameter `MonteCarloLevel` is set to a small positive integer s, then a probabilistic Monte-Carlo modular technique is used. Rather than using sufficient primes to cover the Hadamard bound divided by the divisor d, this version of the algorithm terminates when the constructed residue remains constant for s steps. The probability of this being wrong is non-zero but extremely small, even if s is only 1 or 2. If the level is set to 0, then the normal deterministic algorithm is used. Setting the parameter `Proof` to `false`is equivalent to setting `MonteCarloLevel` to 2.

If the coefficient ring is $\mathbf{Z}$ and the parameter `Divisor` is set to an integer d, then d must be a known exact divisor of the determinant (the sign does not matter), and the algorithm may be sped up because of this knowledge.

```
Trace(a)
```

Given an element a of a subalgebra of $M_n(S)$, return the trace of a as an element of S.

```
Transpose(a)
```

Given an element a of a subalgebra of $M_n(S)$, return the transpose of a as an element of $M_n(S)$.

`Order(a)`

Given an invertible matrix a over any commutative ring, determine the order of a. If a has infinite order, the function may become stuck indefinitely since it cannot prove such.

`FactoredOrder(a)`

Given an invertible matrix a over a finite field, return the order of a in factored form.

`ProjectiveOrder(a)`

Given an invertible matrix a over a finite field, return the projective order o of a and a scalar s such that $a^o = sI$.

`FactoredProjectiveOrder(a)`

Given an invertible matrix a over a finite field, return the projective order o of a in factored form and a scalar s such that $a^o = sI$.

`CharacteristicPolynomial(a:` *parameters*`)`

`Al` MonStgElt *Default : "Modular"*

`Proof` BoolElt *Default :* `true`

The characteristic polynomial of the element a belonging to the algebra $M_n(R)$, where R can be any commutative ring. The parameter `Al` may be used to specify the algorithm used. The algorithm `Modular` (the default) can be used for matrices over Z and Q—in such a case the parameter `Proof` can also be used to suppress proof of correctness. The algorithm `Hessenberg`, allowed for matrices over fields, works by first reducing the matrix to Hessenberg form. The algorithm `Interpolation`, allowed for matrices over Z and Q, works by evaluating the characteristic matrix of a at various points and then interpolating. The algorithm `Trace`, allowed for matrices over fields, works by calculating the traces of powers of a.

`MinimalPolynomial(a)`

The minimal polynomial of the element a belonging to the module $M_n(R)$, where R is a field or Z.

`HessenbergForm(a)`

The Hessenberg form for the matrix a belonging to the algebra $M_n(K)$, where the coefficient ring K must be a field. The form has zero entries above the super-diagonal. (This form is used in one of the characteristic polynomial algorithms.)

`Adjoint(a)`

The adjoint of the matrix a belonging to the algebra $M_n(K)$, where the coefficient ring K must be a ring with exact division whose characteristic must be zero or greater than the degree of a.

`Eigenvalues(a)`

The eigenvalues of the matrix a returned as a set of pairs, each of which gives the value of a distinct eigenvalue and its multiplicity. The coefficient ring must have a polynomial roots algorithm.

`Eigenspace(a, e)`

The eigenspace of the matrix a, corresponding to the eigenvalue e, returned as a submodule of the base module for the parent algebra of a (i.e. the kernel of $a - eI$). If the ring element e is not a eigenvalue for the matrix a then the trivial space is returned.

83.8 Elements of M_n as Homomorphisms

The matrix algebra $M_n(S)$ may also be viewed as the module $\mathrm{Hom}(S^{(n)}, S^{(n)})$. At present this will not happen automatically so that in order to treat elements of $M_n(S)$ as homomorphisms, it is necessary to explicitly coerce the matrix into $\mathrm{Hom}(S^{(n)}, S^{(n)})$. However, two fundamental homomorphism-type operators are provided for elements of $M_n(S)$.

`Image(a)`

`RowSpace(a)`

Given an element of $M_n(S)$, return the image of the module $S^{(n)}$ under the homomorphism represented by the matrix a (as an element of $S^{(n)}$).

`Kernel(a)`

`NullSpace(a)`

`Al` MonStgElt *Default : "Default"*

Given an element of $M_n(S)$, return the kernel of the homomorphism represented by the matrix a (as an element of $S^{(n)}$).

`RowNullSpace(a)`

`NullspaceOfTranspose(a)`

Given an element of $M_n(S)$, return the row nullspace of the homomorphism represented by the matrix a (as an element of $S^{(n)}$). This is equal to the kernel of the transpose of a.

83.9 Elementary Operations on Subalgebras and Ideals

83.9.1 Bases

The functions described here assume that the matrix algebra R is defined over a ring S with a matrix echelonization algorithm. Magma computes a basis for R considered as a S-module when necessary so then operations like membership testing can be performed. The following functions allow one to access this basis.

```
Dimension(R)
```

Assuming that R is a subalgebra of $M_n(S)$, return the dimension of R, considered as a S-module.

```
Basis(R)
```

Assuming that R is a subalgebra of $M_n(S)$, return the S-basis of R, considered as a S-module. The basis is returned as a sequence of matrices of R.

```
BasisElement(R, i)
```

Given R a subalgebra of $M_n(S)$, return the i-th element of the S-basis of R, where i must be between 1 and the dimension of R.

```
Coordinates(R, X)
```

Assuming that R is a subalgebra of $M_n(S)$, and given an element X of R, return the coordinates of X with respect to the basis of R. If R has dimension k over its coefficient ring S, and R has basis $U_1, \ldots, U_k$, the coordinates are returned as the unique sequence $[a_1, \ldots, a_k]$ of elements of S such that $X = a_1U_1 + \ldots + a_rU_r$.

83.9.2 Intersection of Subalgebras

```
R meet T
```

Given algebras R and S that are subalgebras of the same complete algebra $M_n(S)$, where S is a PIR, this operator constructs their intersection.

83.9.3 Membership and Equality

The operations described here assume that the matrix algebra is defined over a principal ideal ring.

```
x in R
X subset R
T subset R
```

Given a matrix x (set of matrices X, matrix algebra T) and a matrix algebra R all belonging to a common matrix algebra defined over a PIR, return `true` if x (X, T, respectively) is contained in R, `false` otherwise.

```
x notin R
```

```
X notsubset R
```

```
T notsubset R
```

Given a matrix x (set of matrices X, matrix algebra T) and a matrix algebra R all belonging to a common matrix algebra defined over a PIR, return `true` if x (X, T, respectively) is not contained in R, `false` otherwise.

```
R eq T
```

Given a matrix algebra R, and a matrix algebra T, return `true` if R is equal to T, `false` otherwise.

```
R ne T
```

Given a matrix algebra R and a matrix algebra T, return `true` if R is not equal to T, `false` otherwise.

83.10 Accessing and Modifying a Matrix

83.10.1 Indexing

```
a[i]
```

Given an element a belonging to the matrix algebra R over the ring S, return the i-th row of a as an element of the natural S-module associated with R.

```
a[i] := u
```

Given an element a belonging to the matrix algebra R over the ring S, an integer i in the range $[1, n]$ and an element u of the natural S-module associated with R, replace the i-th row of a by the vector u.

```
a[i, j]
```

Given an element a belonging to the matrix algebra R over the ring S, return the (i, j)-th entry of a as an element of S.

```
a[i, j] := t
```

Given an element a belonging to the matrix algebra R over the ring S, integers i and j in the range $[1, n]$, and an element t of S, replace the (i, j)-th entry of a by t.

```
ElementToSequence(a)
```

```
Eltseq(a)
```

Given an element a of the matrix algebra R over S, where $a = (a_{ij})$, $1 \leq i$, $j \leq n$, return a as the sequence of elements of S:

$$[a_{11}, \ldots, a_{1n}, a_{21}, \ldots, a_{2n}, \ldots, a_{n1}, \ldots, a_{nn}].$$

83.10.2 Extracting and Inserting Blocks

```
Submatrix(a, i, j, p, q)
```

```
ExtractBlock(a, i, j, p, q)
```

Given a matrix a belonging to a subalgebra of $M_n(S)$ and integers i, j, p and q satisfying the conditions, $1 \leq i + p \leq m$, $1 \leq j + q \leq n$, create the matrix b consisting of the $p \times q$ submatrix of a whose first entry is the (i, j)-th entry of a. If $p \neq q$, the matrix b is created as an element of $\mathrm{Hom}(P, Q)$, where $\mathrm{Rank}(P) = p$, $\mathrm{Rank}(Q) = q$. Otherwise it is created as an element of $M_p(S)$.

```
InsertBlock(~a, b, i, j)
```

(Procedure.) Given that the matrix a belongs to a subalgebra of $M_n(S)$ and the $p \times q$ matrix b is also over S, the integers i, j, p and q must satisfy the conditions, $1 \leq i + p \leq m$, $1 \leq j + q \leq n$. This procedure modifies a so that the $p \times q$ block beginning at the (i, j)-th entry of a is replaced by b.

83.10.3 Joining Matrices

```
HorizontalJoin(X, Y)
```

Given matrices X with r rows and c columns, and Y with r rows and d columns, both over the same coefficient ring R, return the matrix over R with r rows and $(c + d)$ columns obtained by joining X and Y horizontally (placing Y to the right of X).

```
HorizontalJoin(Q)
```

Given a sequence Q of matrices, each having the same number of rows and being over the same coefficient ring R, return the matrix over R obtained by joining the elements of Q horizontally in order.

```
VerticalJoin(X, Y)
```

Given matrices X with r rows and c columns and Y with s rows and c columns, both over the same coefficient ring R, return the matrix with $(r + s)$ rows and c columns over R obtained by joining X and Y vertically (placing Y underneath X).

```
VerticalJoin(Q)
```

Given a sequence Q of matrices, each having the same number of columns and being over the same coefficient ring R, return the matrix over R obtained by joining the elements of Q vertically in order.

```
DiagonalJoin(X, Y)
```

Given matrices X with a rows and b columns and Y with c rows and d columns, both over the same coefficient ring R, return the matrix with $(a+c)$ rows and $(b+d)$ columns over R obtained by joining X and Y diagonally (placing Y diagonally to the right of and underneath X, with zero blocks above and below the diagonal).

```
DiagonalJoin(Q)
```

Given a sequence Q of matrices, each being over the same coefficient ring R, return the matrix over R obtained by joining the elements of Q diagonally in order.

83.10.4 Row and Column Operations

For the following operations, a is an element of a subring of the matrix algebra $M_n(S)$, u is a non-zero element of S, and i and j are integers in the range $[1, n]$. Each of the operations described here acts on the matrix in place, and is therefore implemented as a procedure.

```
SwapRows(~a, i, j)
```

Mutate the matrix a by interchanging rows i and j.

```
MultiplyRow(~a, u, j)
```

Mutate the matrix a by multiplying row j by the scalar u.

```
AddRow(~a, u, i, j)
```

Mutate the matrix a by adding u times row i to row j.

```
SwapColumns(~a, i, j)
```

Mutate the matrix a by interchanging columns i and j.

```
MultiplyColumn(~a, u, i)
```

Mutate the matrix a by multiplying column i by the scalar u.

```
AddColumn(~a, u, i, j)
```

Mutate the matrix a by adding u times column i to column j.

83.11 Canonical Forms

83.11.1 Canonical Forms for Matrices over Euclidean Domains

The functions given here apply to matrices defined over Euclidean Domains. See also the section on Reduction in the Lattices chapter for a description of the function LLL and related functions.

```
EchelonForm(a)
```

The (row) echelon form of matrix a belonging to a submodule of the module $M_n(S)$. This function returns two values:

(a) The (row) echelon form e of a; and

(b) A matrix b such that $b * a = e$, i.e. b is a product of elementary matrices that transforms a into echelon form.

```
ElementaryDivisors(a)
```

The elementary divisors of the matrix a belonging to a submodule of the module $M_n(S)$. The divisors are returned as a sequence $[e_1, \ldots, e_d]$, $e_i|e_{i+1}$ $(i = 1, \ldots, d-1)$ of d elements of R (which may include ones), where d is the rank of a. If R is a field, the result is always the sequence of r ones, where r is the rank of a.

```
HermiteForm(X)
```

`Al`	MonStg	*Default :* "LLL"
`Optimize`	BoolElt	*Default :* `true`
`Integral`	BoolElt	*Default :* `true`

The row Hermite normal form of an matrix X belonging to the matrix algebra $M_n(R)$. The coefficient ring R must be an Euclidean domain. This function returns two values:

(a) The Hermite normal form H of X; and

(b) A unimodular matrix T such that $T \cdot X = H$, i.e., T is the product of elementary matrices which transforms X into Hermite normal form.

If R is the ring of integers **Z** and the matrix T is requested (i.e., if an assignment statement is used with two variables on the left side), then the LLL algorithm will be used by default to improve T (using the kernel of X) so that the size of its entries are very small. If the parameter `Optimize` is set to `false`, then this will not happen (which will be faster but the entries of T will not be as small). If the parameter `Integral` is set to `true`, then the integral (de Weger) LLL method will be used in the LLL step, instead of the default floating point method. The integral method will often be faster if the rank of the kernel of X is very large (say 200 or more).

If R is the ring of integers **Z** and the parameter `Al` is set to the string `"Sort"`, then the sorting-gcd algorithm will be used. However, the new algorithm will practically always perform better than the sorting-gcd algorithm.

```
SmithForm(a)
```

The Smith normal form for the matrix a belonging to a submodule of the module $M_n(S)$, where S is a Euclidean Domain. This function returns three values:

(a) The Smith normal form s of a; and

(b) Unimodular matrices b and c such that $b * a * c = s$, i.e. b and c are matrices that transform a into Smith normal form.

Example H83E6

We illustrate some of these operations in the context of the algebra $M_4(K)$, where K is the field $\mathbf{F}_8$.

```
> K<w> := FiniteField(8);
> M := MatrixAlgebra(K, 4);
> A := M ! [1, w, w^5, 0,  w^3, w^4, w, 1,  w^6, w^3, 1, w^4, 1, w, 1, w ];
> A;
[  1   w w^5   0]
[w^3 w^4   w   1]
[w^6 w^3   1 w^4]
[  1   w   1   w]
> EchelonForm(A);
[1 0 0 0]
[0 1 0 0]
[0 0 1 0]
[0 0 0 1]
```

83.11.2 Canonical Forms for Matrices over a Field

The functions in this group apply to elements of matrix algebras whose coefficient rings are fields which allow factorization of univariate polynomials over them.

```
PrimaryRationalForm(a)
```

The primary rational canonical form of a matrix a belonging to $M_n(K)$, where the coefficient ring K must be a field allowing factorization of univariate polynomials over it. Each block corresponds to a power of an irreducible polynomial. This function returns three values:

(a) The primary rational canonical form p of a;

(b) A non-singular matrix t such that $t * a * t^{-1} = p$;

(c) A sequence of pairs corresponding to the blocks of p where each pair consists of the irreducible polynomial and multiplicity making up the block.

```
JordanForm(a)
```

The (generalized) Jordan canonical form for the matrix a belonging to the algebra $M_n(K)$, where the coefficient ring K must be a field allowing factorization of univariate polynomials over it. This function returns three values:

(a) The (generalized) Jordan canonical form j of a;

(b) A non-singular matrix t such that $t * a * t^{-1} = j$;

(c) A sequence of pairs corresponding to the blocks of j where each pair consists of the irreducible polynomial and multiplicity making up the block.

`RationalForm(a)`

The rational canonical form of a matrix a belonging to $M_n(K)$, where the coefficient ring K must be a field allowing factorization of univariate polynomials over it. For each block before the last block, the polynomial corresponding to that block divides the polynomial corresponding to the next block. This function returns three values:

(a) The rational canonical form f of a;

(b) A non-singular matrix t such that $t * a * t^{-1} = f$;

(c) A sequence containing the polynomials corresponding to each block (each non-last one dividing the next).

`PrimaryInvariantFactors(a)`

The primary invariant factors of the matrix a. This is the same as the third return value of `PrimaryRationalForm(a)` or `JordanForm(a)`. The coefficient ring must be a field allowing factorization of univariate polynomials over it.

`InvariantFactors(a)`

The invariant factors of the matrix a. This is the same as the third return value of `RationalForm(a)`. The coefficient ring must be a field allowing factorization of univariate polynomials over it.

`IsSimilar(a, b)`

Returns `true` iff the matrix a is similar to the matrix b. If a is similar to b, a transformation matrix t is also returned with $t * a * t^{-1} = b$. The coefficient ring must be a field allowing factorization of univariate polynomials over it.

Example H83E7

We consider the algebra $M_5(P)$, where P is the polynomial ring in indeterminate x over the field $\mathbf{F}_5$. We take the matrix having $x^i + x^j$ in its (i, j)-th position.

```
> K := GaloisField(5);
> P<x> := PolynomialAlgebra(K);
> M := MatrixAlgebra(P, 5);
> a := M ! [x^i + x^j: i, j in [1..5]];
> a;
[      2*x   x^2 + x   x^3 + x   x^4 + x   x^5 + x]
[  x^2 + x     2*x^2 x^3 + x^2 x^4 + x^2 x^5 + x^2]
[  x^3 + x x^3 + x^2     2*x^3 x^4 + x^3 x^5 + x^3]
[  x^4 + x x^4 + x^2 x^4 + x^3     2*x^4 x^5 + x^4]
[  x^5 + x x^5 + x^2 x^5 + x^3 x^5 + x^4     2*x^5]
> ElementaryDivisors(a);
[
    x,
    x^3 + 3*x^2 + x
]
> Rank(a);
```

```
2
```

Example H83E8

We construct a 5 by 5 matrix over the finite field with 5 elements and then calculate various canonical forms. We verify the correctness of the polynomial invariant factors corresponding to the rational form by calculating the Smith form of the characteristic matrix of the original matrix.

```
> K := GF(5);
> P<x> := PolynomialRing(K);
> A := MatrixAlgebra(K, 5);
> a := A !
> [
>     0, 2, 4, 2, 0,
>     2, 2, 2, 3, 3,
>     3, 4, 4, 1, 3,
>     0, 0, 0, 0, 1,
>     0, 0, 0, 1, 0
> ];
> a;
[0 2 4 2 0]
[2 2 2 3 3]
[3 4 4 1 3]
[0 0 0 0 1]
[0 0 0 1 0]
> PrimaryInvariantFactors(a);
[
    <x + 1, 1>,
    <x + 1, 1>,
    <x + 4, 1>,
    <x + 4, 1>,
    <x + 4, 1>
]
> r, t, f := RationalForm(a);
> r;
[1 0 0 0 0]
[0 0 1 0 0]
[0 1 0 0 0]
[0 0 0 0 1]
[0 0 0 1 0]
> t;
[1 3 0 2 1]
[2 1 2 2 0]
[3 4 3 4 1]
[1 0 0 0 0]
[0 2 4 2 0]
> f;
[
    x + 4,
    x^2 + 4,
    x^2 + 4
]
> PA := MatrixAlgebra(P, 5);
> ax := PA ! x - PA ! a;
> ax;
[    x     3     1     3     0]
[    3 x + 3     3     2     2]
[    2     1 x + 1     4     2]
[    0     0     0     x     4]
[    0     0     0     4     x]
> SmithForm(ax);
[      1       0       0       0       0]
[      0       1       0       0       0]
[      0       0   x + 4       0       0]
[      0       0       0 x^2 + 4       0]
[      0       0       0       0 x^2 + 4]
> ElementaryDivisors(ax);
[
    1,
    1,
    x + 4,
    x^2 + 4,
    x^2 + 4
]
```

83.12 Diagonalising Commutative Algebras over a Field

The functions in this group apply to (elements of) matrix algebras whose coefficient rings are fields which allow the construction of splitting fields of univariate polynomials over them. Specifically, the base field must be the rationals, a number field, a finite field, or an algebraically closed field.

`CommonEigenspaces(Q)`

The common eigenspaces of the sequence Q of pairwise-commuting matrices. If the first sequence returned is V and the second is E, then $E[i][j]$ is the eigenvalue of $Q[i]$ corresponding to the common eigenspace $V[j]$.

`CommonEigenspaces(A)`

The common eigenspaces of the commutative matrix algebra A. If the first sequence returned is V and the second is E, then $E[i][j]$ is the eigenvalue of $A.i$ corresponding to the common eigenspace $V[j]$.

`Diagonalisation(Q)`

`Diagonalization(Q)`

The diagonalisation of the sequence Q of pairwise-commuting matrices. That is, a sequence of diagonal matrices of the form $[P * Q[1] * P^{-1}, P * Q[2] * P^{-1}, \ldots]$. The second value returned is the matrix P. Note that the returned values may have a larger base field than the input.

`Diagonalisation(A)`

`Diagonalization(A)`

The diagonalisation of the commutative matrix algebra A. That is, an algebra with diagonal generators $[P * A.1 * P^{-1}, P * A.2 * P^{-1}, \ldots]$. The second value returned is the matrix P.

Example H83E9

```
> M := MatrixAlgebra(Rationals(),2);
> x := M![0,1,-2,0];
> y := M![0,3,-6,0];
> CommonEigenspaces([x,y]);
[*
    Vector space of degree 2, dimension 1 over Number Field with defining
    polynomial $.1^2 + 2 over the Rational Field
    Generators:
    (      1 1/2*r.1)
    Echelonized basis:
    (      1 1/2*r.1),
    Vector space of degree 2, dimension 1 over Number Field with defining
    polynomial $.1^2 + 2 over the Rational Field
    Generators:
    (       1 -1/2*r.1)
    Echelonized basis:
    (       1 -1/2*r.1)
*] [
    [
        -r.1,
        -3*r.1
    ],
    [
        r.1,
        3*r.1
    ]
]
> Diagonalisation(sub<M|x,y>);
Matrix Algebra of degree 2 with 2 generators over Number Field with defining
polynomial $.1^2 + 2 over the Rational Field
```

```
[       1  1/2*r.1]
[       1 -1/2*r.1]
```

83.13 Solutions of Systems of Linear Equations

`IsConsistent(A, w)`

Given a matrix A belonging to $M_n(R)$ and a vector w belonging to the tuple module $R^{(n)}$, return `true` iff the system of linear equations $v * A = w$ is consistent. If the system is consistent, then the function will also return:

(a) A particular solution v;

(b) The kernel K of A so that $(v + k) * A = w$ for $k \in K$.

`IsConsistent(A, W)`

Given a matrix A belonging to $M_n(R)$ and a sequence W of vectors belonging to the tuple module $R^{(m)}$, return `true` iff the system of linear equations $V[i] * A = W[i]$ for each i is consistent. If the systems are all consistent, then the function will also return:

(a) A solution sequence V;

(b) The kernel K of A so that $(V[i] + k) * A = W[i]$ for $k \in K$.

`Solution(A, w)`

Given a matrix A belonging to $M_n(R)$ and a vector w belonging to the tuple module $R^{(n)}$, solve the system of linear equations $v*A = w$. The function returns two values:

(a) A particular solution v;

(b) The kernel K of A so that $(v + k) * A = w$ for $k \in K$.

`Solution(A, W)`

Given a matrix A belonging to $M_n(R)$ and a sequence W of vectors belonging to the tuple module $R^{(n)}$, solve the system of linear equations $V[i] * A = W[i]$ for each i. The function returns two values:

(a) A solution sequence V;

(b) The kernel K of A so that $(V[i] + k) * A = W[i]$ for $k \in K$.

83.14 Presentations for Matrix Algebras

Magma has the capability of constructing a presentation for an algebra A generated by a collection $\alpha_1, \ldots, \alpha_t$ of $n \times n$ matrices over a finite field k. The presentation has the form

$$A \cong P/I$$

where P is a free algebra in noncommuting variable and I is a two-sided ideal in P. The presentation is obtained by computing a set of primitive idempotents for the algebra and extracting generators for the radical of the algebra. For such an algebra there is a sequence

$$0 \longrightarrow \mathrm{Rad}(A) \longrightarrow A \longrightarrow A/\mathrm{Rad}(A) \longrightarrow 0$$

which is split in the sense that $A/\mathrm{Rad}(A)$ is a subalgebra A. Moreover, $A/\mathrm{Rad}(A)$ is a direct sum of complete matrix algebras A_i over extensions K_i of the field k. Each complete matrix algebra A_i is generated by two elements b_i and t_i. The element b_i has minimal rank and $b_i^{n_i-1} = e_i$ is a primitive idempotent where n_i is the number of elements in the field K_i. That is, b_i is a generator for the multiplicative group of nonzero elements of $e_iA_ie_i \cong K_i$. The element t_i is conjugate to a permutation matrix of degree n_i in A_i. The elements $z_1, \ldots, z_s$ are generators of the radical of A. These elements are computed so as to lie in the condensed algebra eAe where $e = \sum e_i$.

The calculation produces also a set of generators and relations for the condensed algebra eAe. This algebra is Morita equivalent to A, and hence shares many of the same homological properties of the algebra A.

In the course of obtaining the presentation, several aspects of the algebra are computed.

83.14.1 Quotients and Idempotents

```
NaturalFreeAlgebraCover(A)
```

Returns the map of a free algebra onto the matrix algebra A, such that the variables of the free algebra go to the generators of A.

```
SimpleQuotientAlgebras(A)
```

The simple quotient algebras of the matrix algebra A. The output is a record having the following fields:

(a) The actual simple algebras that are the quotients of the algebra A. The function returns the sequence of mappings from the natural free-algebra cover of A to the quotients. The variable corresponding to a generator of A is mapped to the corresponding generator of the quotient. (field name `SimpleQuotients`)

(b) The degrees of the quotients as matrix algebras over their centers (field name `DegreesOverCenters`).

(c) The degrees of the extension of center of the quotient algebra over the base field of A (field name `DegreesOfCenters`).

(d) The number of elements in the center of the quotient algebra (field name `OrdersOfCenters`).

`PrimitiveIdempotentData(A)`

The initial data for a decomposition of the matrix algebra A. The output is a sequence of records, one for each simple quotient algebra of A, each consisting of the following fields.

(a) `AlgebraIdempotent` : An idempotent whose image in the simple quotient is the identity matrix.

(b) `PrimitiveIdempotent` : A primitive idempotents whose image in the quotient is primitive.

(c) `PrimitiveIdempotentOnQuotient` : The image of the primitive idempotent in the quotient algebra.

(d) `FieldGenerator` : A multiple of the primitive idempotent which is a field generator for the center of the algebra.

(e) `FieldGeneratorOnQuotient` : The image of the field generator in the quotient algebra.

(f) `GeneratingPolForCenter` : The minimal polynomial for the matrix of the field generator.

`PrimitiveIdempotents(A)`

A list of primitive idempotent for the matrix algebra A, one idempotent for each irreducible module.

`RanksOfPrimitiveIdempotents(A)`

The sequence of ranks of the primitive idempotents for the matrix algebra A.

`NaturalFreeAlgebraCover(A)`

Returns the map of a free algebra onto the matrix algebra A, such that the variables of the free algebra go the generators of the algebra.

`CondensedAlgebra(A)`

Returns the algebra eAe where e is a sum of primitive idempotents, one for each simple A-module.

Example H83E10

We form a matrix algebra over the field with three elements generated by two elements. The algebra is block upper triangular, where the upper left block is a field extension of degree three and the lower block is a filed extension of degree 2.

```
> a1 := KMatrixSpace(GF(3),3,3)![0,1,0,0,0,1,-1,0,1];
> a2 := KMatrixSpace(GF(3),2,2)![0,1,-1,0];
> z1 := KMatrixSpace(GF(3),5,5)!0;
> z2 := InsertBlock(z1,a1,1,1);
> z2;
[0 1 0 0 0]
```

```
[0 0 1 0 0]
[2 0 1 0 0]
[0 0 0 0 0]
[0 0 0 0 0]
> z3 := InsertBlock(z1,a2,4,4);
```

$z2$ and $z3$ are the matrices of the field extensions. Next, add an entry to $z3$ that is an extension class between the two algebras.

```
> z3[1][4] := 1;
> z3;
[0 0 0 1 0]
[0 0 0 0 0]
[0 0 0 0 0]
[0 0 0 0 1]
[0 0 0 2 0]
> A := MatrixAlgebra<GF(3),5|z2,z3>;
```

We can check to see if the quotients came out as expected.

```
> SimpleQuotientAlgebras(A);
rec<recformat<SimpleQuotients: SeqEnum, DegreesOverCenters:
SeqEnum, DegreesOfCenters: SeqEnum, OrdersOfCenters: SeqEnum> |
    SimpleQuotients := [
        Mapping from: Free associative algebra of rank 2 over
                GF(3) to Matrix Algebra of degree 3 with 2
            generators over GF(3),
        Mapping from: Free associative algebra of rank 2 over
                GF(3) to Matrix Algebra of degree 2 with 2
            generators over GF(3)
    ],
    DegreesOverCenters := [ 1, 1 ],
    DegreesOfCenters := [ 3, 2 ],
    OrdersOfCenters := [ 27, 9 ]
```

Here are the idempotents.

```
> PrimitiveIdempotents(A);
[
    [1 0 0 0 0]
    [0 1 0 0 0]
    [0 0 1 0 2]
    [0 0 0 0 0]
    [0 0 0 0 0],
    [0 0 0 0 0]
    [0 0 0 0 0]
    [0 0 0 0 1]
    [0 0 0 1 0]
    [0 0 0 0 1]
]
```

Finally we have the Cartan matrix.

```
> CartanMatrix(A);
[1 3]
[0 1]
```

83.14.2 Generators and Presentations

We use the idempotents as the first step in the presentation process. The generators of the algebra consist of the field generators, each of which is a generator of the center of a simple quotient algebra multiplied by a corresponding primitive idempotent, permutation matrices, one for each simple quotient algebra and generators for the radical, all of which can be taken to be in the subalgebra eAe where e is the sum of one primitive idempotent for each simple quotient algebra.

For display purposes the variables for the free algebra of the presentations are given names. The variables $b.i$ represent field generators, while the $t.i$'s are permutation matrices and the $z.i$'s are the generators of the radical. The reader should be warned that these names are not suitable for input into other function.

```
SemisimpleGeneratorData(A)
```

The data on the semisimple generators of the algebra A, that is the generators in A of $A/Rad(A)$. Some of the output is intended for use in other functions. The return is a sequence of records, one for each simple quotient algebra. Each record consists of the following fields.

(a) `PowersOfFieldGenerators` : A record consisting of look-up tables for the powers of the field generators as elements both of the algebra A and the simple quotient algebra.

(b) `Permutation` : The permutation matrix of the quotient algebra as an element of A.

(c) `PermutationOnQuotient` : The permutation matrix on the quotient.

(d) `FieldGenerator` : The matrix of the field generator as an element of A.

(e) `FieldGeneratorOnQuotient` : The matrix of the field generator as an element of the quotient algebra.

(f) `PrimitiveIdempotent` : The matrix of the primitive idempotent on A.

(g) `PrimitiveIdempotentOnQuotient` : The matrix of the primitive idempotent on the quotient algebra.

(h) `GeneratingPolForCenter` : The Galois polynomial for the extension of the center of the quotient algebra over the base ring.

`AlgebraGenerators(A)`

The standard generators of the matrix algebra A. The output is a record consisting of the following fields.

(a) `FieldGenerators` : The sequence of matrices of the field generators, one for each simple quotient algebra.

(b) `PermutationMatrices` : The sequence of permutation matrices for the quotient algebras as elements of A, one for each quotient algebra.

(c) `PrimitiveIdempotents` : The sequence of primitive idempotents for A, one for each quotient algebra.

(d) `RadicalGenerators` : A set of generators for the radical of A arranged as a list of lists of generators of the radical of e_iAe_j for e_i and e_j primitive idempotents.

(e) `SequenceRadicalGenerators` : A sequence of minimal generators of radical of A.

(f) `GeneratingPolynomialsForCenters` : The galois polynomials of the centers over the base field.

(g) `StandardFormConjugationMatrices` : The matrices which conjugate the element of A into the standard form relative to the computed primitive idempotents for A.

`AlgebraStructure(A)`

The accumulated structure of the matrix algebra A. The return is a record with the following fields. Some of this information is saved to be used in other calculations.

(a) `FreeAlgebra` : The free algebra in the newly computed variables for the algebra.

(b) `RelationsIdeal` : The ideal of relations among the new computed generators.

(c) `StandardFreeAlgebraCover` : The map from the free algebra to A.

(d) `FieldGenerators` : The matrices in A of the generators of the centers of the simple quotient algebras of A.

(e) `PermutationMatrices` : The permutation matrices each of which together with the corresponding field generator, generates a simple quotient algebra as a subalgebra of A.

(f) `PrimitiveIdempotents` : The primitive idempotents, each of which is a power of the corresponding field generator.

(g) `RadicalGenerators` : A list of lists giving the generators of the radical of A which are in e_iAe_j where $\{e_i\}$ are the primitive idempotents.

(h) `CondensedRadicalBasis` : The condensed matrices in eAe of a basis for the radical of the algebra. The output is a list of lists giving the basis for e_iAe_j. Each entry in the list of lists is a tuple consisting of a matrix in the condensed algebra and the monomial which expresses this matrix as a product of the generators.

(i) `CondensedFieldGenerators` : The condensed matrices in eAe of the field generators.

(j) `FieldPolynomials` : The sequence of minimal polynomials of the field generators.

(k) `DegreesOfSimpleModules` : The dimensions of the Simple modules of A.

(l) `DegreeOfFieldExtensions` : The degrees of the centers of simple quotient algebras of A over the base field of A.

(m) `SimpleQuotientAlgebras` : The simple quotient algebras of A.

(n) `StandardFormConjugationMatrices` : The matrices which conjugate the algebra A into standard form with respect to the computed system of primitive idempotents.

`Presentation(A)`

The presentation in generators and relations of the matrix algebra A. The function returns the free algebra and the relations ideal calculated in the algebra structure program, as well as the map from the free algebra to A.

`StandardFormConjugationMatrices(A)`

Returns the pair (M and M^{-1}) of matrices that conjugate the matrix algebra A into standard form with respect to a chosen set of primitive idempotents.

`CondensationMatrices(A)`

The matrices, conjugating by which, gives the condensation of A.

`SequenceOfRadicalGenerators(A)`

The sequence of matrices of elements that generate the radical of A.

`CartanMatrix(A)`

The Cartan Matrix of the algebra A.

Example H83E11

In this example we form the permutation module M of the symmetric group G on seven letters by the Young subgroup that is the direct product H of two copies of the symmetric group on three letters. The algebra A is the image of the group algebra in the ring of endomorphisms of M.

```
> G := Sym(7);
> H := sub<G| G!(1,2,3),G!(1,2),G!(4,5,6), G!(4,5)>;
> M := PermutationModule(G, H, GF(5));
> M;
GModule M of dimension 140 over GF(5)
> A := Action(M);
```

We see that A has seven simple quotients.

```
> SimpleQuotientAlgebras(A);
```

```
rec<recformat<SimpleQuotients: SeqEnum, DegreesOverCenters:
SeqEnum, DegreesOfCenters: SeqEnum, OrdersOfCenters: SeqEnum> |
    SimpleQuotients := [
        Mapping from: Free associative algebra of rank 2 over
                GF(5) to Matrix Algebra of degree 35 with 2
            generators over GF(5),
        Mapping from: Free associative algebra of rank 2 over
                GF(5) to Matrix Algebra of degree 15 with 2
            generators over GF(5),
        Mapping from: Free associative algebra of rank 2 over
                GF(5) to Matrix Algebra of degree 13 with 2
            generators over GF(5),
        Mapping from: Free associative algebra of rank 2 over
                GF(5) to Matrix Algebra of degree 8 with 2
            generators over GF(5),
        Mapping from: Free associative algebra of rank 2 over
                GF(5) to Matrix Algebra of degree 8 with 2
            generators over GF(5),
        Mapping from: Free associative algebra of rank 2 over
                GF(5) to Matrix Algebra of degree 6 with 2
            generators over GF(5),
        Mapping from: Free associative algebra of rank 2 over
                GF(5) to Matrix Algebra of degree 1 with 2
            generators over GF(5)
    ],
    DegreesOverCenters := [ 35, 15, 13, 8, 8, 6, 1 ],
    DegreesOfCenters := [ 1, 1, 1, 1, 1, 1, 1 ],
    OrdersOfCenters := [ 5, 5, 5, 5, 5, 5, 5 ]
    >
> RanksOfPrimitiveIdempotents(A);
[ 1, 1, 3, 2, 1, 4, 3 ]
```

The condensed algbra of A is a much smaller object.

```
> B := CondensedAlgebra(A);
> B;
Matrix Algebra of degree 15 with 13 generators over GF(5)
> CartanMatrix(A);
[1 0 0 0 0 0 0]
[0 1 0 0 0 0 0]
[0 0 2 0 1 0 1]
[0 0 0 1 0 1 0]
[0 0 1 0 1 0 0]
[0 0 0 1 0 2 0]
[0 0 1 0 0 0 2]
```

From the Cartan Matrix we can see that A has four blocks. The first and second simple subalgebras are in blocks by themselves. The subalgebras numbers 3, 5, and 7 form another block as do the subalgebras 4 and 6. Note that B, being Morita equivalent to A, has the same Cartan Matrix and the same block structure.

83.14.3 Solving the Word Problem

The presentation machinery also gives a test for membership in a matrix algebra. If A is a subalgebra of the $n \times n$ matrices generated by some collection of matrices, Magma can tell if any $n \times n$ matrix is an element of A. If the element is in A then Magma can write the element as a polynomial in the polynomial ring of the presentation.

```
WordProblemData(A)
```

The data needed for the solution to the word problem. The output is a list of lists of basis elements for the radical together with the corresponding monomials in the generators of the free algebra.

```
WordProblem(A, x)
```

Returns `true` if the matrix x is in the subalgebra A, and if true returns also an expression of the element x as a polynomial in the presentation of A.

Example H83E12

In this example we form the permutation module for the symmetric group G acting on the coset space of the normalizer H of Sylow 3-subgroup of G. The coefficients are in the field with two elements. The algebra A is the image of the group algebra in the endomorphism ring of the permutation module.

```
> G := Sym(5);
> H := Normalizer(G,Sylow(G,3));
> M := PermutationModule(G,H, GF(2));
> M;
GModule M of dimension 10 over GF(2)
> A := Action(M);
```

Here we get the presentation of A.

```
> P, I, mu  := Presentation(A);
> Dimension(P/I);
42
```

Thus the dimension of A is 42.

```
> CartanMatrix(A);
[1 0 1]
[0 1 0]
[1 0 2]
> B := CondensedAlgebra(A);
> Q, J, theta := Presentation(B);
```

The presentation of B has the form:

```
> J;
```

```
Two-sided ideal of Free associative algebra of rank 5 over GF(2)
Non-commutative Graded Lexicographical Order
Variables: b_1, b_2, b_3, z_1, z_2
Groebner basis:
[
    b_2^2 + b_2,
    b_2*b_3,
    b_2*z_1,
    b_2*z_2,
    b_3*b_2,
    b_3^2 + b_3,
    b_3*z_1,
    b_3*z_2 + z_2,
    z_1*b_2,
    z_1*b_3 + z_1,
    z_1^2,
    z_1*z_2,
    z_2*b_2,
    z_2*b_3,
    z_2^2,
    b_1 + b_2 + b_3 + 1
]
```

The matrix $A.1$ is the first of the original generators for A. We can check that $A.1$ is an element of the algebra generated by the computed generators of A.

```
> boo, y := WordProblem(A,A.1);
> boo;
true
```

The element y is the expression of $A.1$ as a polynomial in the free algebra P in the new computed generators for A.

```
> y;
t_1^4*b_1*t_1^3 + t_2^4*b_2*t_2^3 + t_1^4*b_1*t_1^2 +
    t_2^3*b_2*t_2^3 + t_1^2*b_1*t_1^3 + t_1^4*b_1*t_1 +
    t_2^2*b_2*t_2^3 + t_2^3*b_2*t_2^2 + t_2^4*b_2*t_2 +
    t_1^2*b_1*t_1^2 + t_1^3*b_1*t_1 + t_1^4*b_1 + t_1^4*z_1 +
    t_2*b_2*t_2^3 + t_2^2*b_2*t_2^2 + t_2^3*b_2*t_2 +
    t_1*b_1*t_1^2 + t_1^3*b_1 + t_3*z_2*t_1^2 + t_1^2*b_1 +
    t_2*b_2*t_2 + t_2^2*b_2 + t_1*b_1 + t_2*b_2 + t_3*b_3 +
    t_3*z_2
```

The mapping mu is the function from the free algebra P into A.

```
> mu(y);
[0 1 0 0 0 0 0 0 0 0]
[0 0 0 1 0 0 0 0 0 0]
[0 0 0 0 1 0 0 0 0 0]
[0 0 0 0 0 1 0 0 0 0]
[0 0 0 0 0 0 1 0 0 0]
```

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

The ultimate check of the accuracy of the computation is that the polynomial expressions give back the original generators.

```
> mu(y) eq A.1;
true
```

Finally we can check whether a random 10×10 matrix is an element of A.

```
> b := Random(Generic(A));
> WordProblem(A,b);
false
```

83.15 Bibliography

[**ABM99**] John Abbott, Manuel Bronstein, and Thom Mulders. Fast Deterministic Computation of Determinants of Dense Matrices. In Sam Dooley, editor, *Proceedings ISSAC'99*, pages 197–204, New York, 1999. ACM Press.

84 GROUP ALGEBRAS

Chapter 84
GROUP ALGEBRAS

84.1 Introduction

Group algebras (or group rings) may be defined over any unital ring R and any group G in which elements have a canonical form (for example permutation groups, matrix groups or polycyclic groups). Basic operations, such as simple arithmetic of elements, may be applied in any such group algebra. Certain functions, however, place stricter requirements on R and G, such as requiring that R have a matrix echelon algorithm in Magma.

84.2 Construction of Group Algebras and their Elements

84.2.1 Construction of a Group Algebra

There are two different representations of group algebra elements used in Magma. Which is most suitable depends on the group G and on the operations required, and must be decided upon when the algebra is created.

The first representation, which requires that G is not too large, is to choose (once and for all) an ordering $(g_1, g_2, \ldots, g_n)$ of the elements of G (where $n = |G|$), and then to store elements of the group algebra $A = R[G]$ as coefficient vectors relative to the basis $(g_1, g_2, \ldots, g_n)$ of A. So an element $a = a_1 * g_1 + a_2 * g_2 + \ldots + a_n * g_n$ $(a_i \in R)$ of A will be stored as the vector $(a_1, a_2, \ldots, a_n)$. This makes for fast arithmetic and allows the use of matrix echelonization for dealing with subalgebras and ideals (if R has a matrix echelon algorithm).

The alternative representation, necessary when dealing with large groups, stores an element $a \in A = R[G]$ as a pair of parallel arrays giving the terms of the element. One array contains the nonzero coefficients of the element and the other contains their associated group elements. This representation allows group algebras to be defined over any group in which the elements have a canonical form, including potentially infinite matrix groups over a ring of characteristic 0 or even free groups. Note however, that operations in such algebras are limited, as the length of the representing arrays may grow exponentially with the number of multiplications performed.

```
GroupAlgebra( R, G: parameters)
```

```
GroupAlgebra<  R, G: parameters >
```

Rep	MonStgElt	*Default :*
Table	BoolElt	*Default :*

Given a unital ring R and a group G (which is not a finitely presented group), create the group algebra $R[G]$. There are two optional arguments associated with this creation function.

The optional parameter Rep can be used to specify which representation should be chosen for elements of the algebra. Its possible values are "Vector" and "Terms". If Rep is not assigned, then MAGMA chooses the vector representation if $|G| \leq 1000$ and the term representation otherwise. If Rep is set to "Vector", MAGMA has to compute the order of G. If it does not succeed in that or the order is too large to construct vectors of this size, then an error message is displayed.

The second optional parameter Table can be used to specify whether or not the multiplication table of the group should be computed and stored upon creation of the algebra. Storing the multiplication table makes multiplication of algebra elements (and all other group algebra operations using this) much faster. Building this table does, however, increase the time needed to create the algebra. The table can only be stored if the vector representation of elements is used. If the Table parameter is not set, then MAGMA stores the multiplication table only if $G \leq 200$.

Example H84E1

We first construct the default group algebra $A = R[G]$ where R is the ring of integers and G is the symmetric group on three points.

```
> G := SymmetricGroup(3);
> R := Integers();
> A := GroupAlgebra( R, G );
> A;
Group algebra with vector representation and stored multiplication table
Coefficient ring: Integer Ring
Group: Permutation group G acting on a set of cardinality 3
    Order = 6 = 2 * 3
        (1, 2, 3)
        (1, 2)
```

Next we construct the group algebra $A = R[G]$ where $R = GF(5)$ and G is the dihedral group of order 100, given as a polycyclic group. This time we specify that the term representation should be used for elements.

```
> G := PCGroup( DihedralGroup(50) );
> A := GroupAlgebra( GF(5), G: Rep := "Terms" );
> A;
Group algebra with terms representation
Coefficient ring: Finite field of size 5
Group: GrpPC : G of order 100 = 2^2 * 5^2
    PC-Relations:
        G.1^2 = Id(G),
        G.2^2 = Id(G),
        G.3^5 = G.4,
        G.4^5 = Id(G),
        G.3^G.1 = G.3^4 * G.4^4,
        G.4^G.1 = G.4^4
```

84.2.2 Construction of a Group Algebra Element

```
elt< A | r, g >
```

Given a group algebra $A = R[G]$, a ring element $r \in R$ and a group element $g \in G$, create the element $r * g$ of A.

```
A ! g
```

Given a group element $g \in G$ create the element $1_R * g$ of the group algebra $A = R[G]$.

```
A ! r
```

Given a ring element $r \in R$, create the element $r * 1_G$ of the group algebra $A = R[G]$.

A ! $[c_1, \ldots, c_n]$

Given a group algebra $A = R[G]$ in vector representation and a sequence $[c_1, \ldots, c_n]$ of $n = |G|$ elements of R, create the element $c_1 * g_1 + \ldots + c_n * g_n$, where $(g_1, \ldots, g_n)$ is the fixed basis of A.

```
Eta(A)
```

For a group algebra $A = R[G]$ in vector representation, create the element $\sum_{g \in G} 1_R * g$ of A.

Example H84E2

We check that $Eta(A)/|G|$ is an idempotent in A (provided the characteristic of R does not divide the order of G).

```
> G := Alt(6);
> A := GroupAlgebra( GF(7), G );
> e := Eta(A) / #G;
> e^2 - e;
0
```

84.3 Construction of Subalgebras, Ideals and Quotient Algebras

Let $A = R[G]$ be a group algebra defined in Magma, where R is a unital ring with a matrix echelon algorithm and G is a group. If A was constructed to use the vector representation for its elements, then subalgebras and left, right and two-sided ideals of A can be constructed. If R is a field, then quotient algebras of A are also possible.

A subalgebra or ideal of a group algebra A is not in general a group algebra; hence, a different type is needed for these objects. Two possibilities are provided in Magma. The first is to construct the subalgebra or ideal to have type `AlgGrpSub`: a subalgebra (which may be an ideal) of a group algebra. Its elements are elements of the group algebra and it remains closely connected to its defining group algebra. By default, ideals of group algebras will be of this type.

The alternative is to create the subalgebra or ideal as an associative algebra given by structure constants (type `AlgAss`). The required structure constants are easily computable from the group algebra. This is the default Magma uses when creating subalgebras of a group algebra.

Regardless of the type given to a subalgebra or ideal, the inclusion map back into the group algebra is also provided.

```
sub< cat : A | L >
```

`cat` Cat *Default :* `AlgGrpSub`

Create the subalgebra S of the group algebra A generated by the elements defined by L, where L is a list of one or more items of the following types:

(a) an element of A;

(b) a set or sequence of elements of A;

(c) a subalgebra of A;

(d) a set or sequence of subalgebras of A;

(e) a sequence of ring elements specifying an element of A, as returned when the `ElementToSequence` (or `Eltseq`) function is applied to the element.

The constructor returns the subalgebra as an element of the category `cat`, where `cat` is either `AlgGrpSub` (the default) or `AlgAss`. As well as the subalgebra itself, the constructor returns the inclusion homomorphism $f : S \to A$.

```
lideal< cat : A | L >
```

`cat` Cat *Default :* `AlgGrpSub`

Create the left ideal of the group algebra A, generated by the elements defined by L, where L is a list as for the `sub` constructor above.

The constructor returns the ideal as an element of the category `cat`, where `cat` is either `AlgGrpSub` (the default) or `AlgAss`. As well as the ideal itself, the constructor returns the inclusion homomorphism $f : I \to A$.

```
rideal< cat : A | L >
```

cat Cat *Default :* `AlgGrpSub`

Create the right ideal of the group algebra A generated by the elements defined by L, where L is a list as for the `sub` constructor above.

The constructor returns the ideal as an element of the category `cat`, where `cat` is either `AlgGrpSub` (the default) or `AlgAss`. As well as the ideal itself, the constructor returns the inclusion homomorphism $f : I \to A$.

```
ideal< cat : A | L >
```

cat Cat *Default :* `AlgGrpSub`

Create the (two-sided) ideal I of the group algebra A generated by the elements defined by L, where L is a list as for the `sub` constructor above.

The constructor returns the ideal as an element of the category `cat`, where `cat` is either `AlgGrpSub` (the default) or `AlgAss`. As well as the ideal itself, the constructor returns the inclusion homomorphism $f : I \to A$.

The above operations can also be applied with a subalgebra (type `AlgGrpSub`) S in place of the group algebra A. In this case the subalgebra and ideal closures are taken in S.

```
a * I
```

For a right ideal I of the group algebra A construct the right ideal $\{a * b : b \in I\}$.

```
I * a
```

For a left ideal I of the group algebra A construct the left ideal $\{b * a : b \in I\}$.

```
quo< A | L >
```

Create the quotient of the group algebra A by the ideal of A generated by the elements defined by L, where L is a list as for the `sub` constructor above.

The constructor returns the quotient as an associative algebra. As well as the quotient Q itself, the constructor returns the natural homomorphism $f : A \to Q$.

```
A / S
```

The quotient of the algebra A by the ideal closure of its subalgebra S.

Example H84E3

We demonstrate how the degrees of the absolutely irreducible characters of a group can be found by computing the Wedderburn decomposition of the group algebra over a suitable finite field. Of course, this is neither the smartest nor the quickest way to get this information. In the example we deal with an extraspecial group of order 3^3.

```
> G := ExtraSpecialGroup(3, 1);
> Exponent(G);
```

```
3
```

We have to choose a finite field that contains the roots of unity that may be required by the absolutely irreducible characters. This is the case for $GF(q)$ if the exponent of G divides $q-1$. In our example, $q=4$ or $q=7$ are possible choices.

```
> FG := GroupAlgebra(GF(4), G);
> FG;
Group algebra with vector representation and stored multiplication table
Coefficient ring: Finite field of size 2^2
Group: Permutation group G acting on a set of cardinality 27
    Order = 27 = 3^3
        (1, 19, 10)(2, 20, 11)(3, 21, 12)(4, 22, 13)(5, 23, 14)(6, 24, 15)(7,
            25, 16)(8, 26, 17)(9, 27, 18)
        (1, 7, 4)(2, 8, 5)(3, 9, 6)(10, 18, 14)(11, 16, 15)(12, 17, 13)(19, 26,
            24)(20, 27, 22)(21, 25, 23)
        (1, 3, 2)(4, 6, 5)(7, 9, 8)(10, 12, 11)(13, 15, 14)(16, 18, 17)(19, 21,
            20)(22, 24, 23)(25, 27, 26)
> MI := MinimalIdeals(FG);
> #MI;
11
```

The minimal ideals are the simple Wedderburn components of the group algebra, corresponding to the absolutely irreducible characters of the group, and their dimensions are the squares of the character degrees.

```
> [ Isqrt(Dimension(I)) : I in MI ];
[ 1, 1, 1, 1, 1, 1, 1, 1, 1, 3, 3 ]
```

Hence, there are 9 linear characters and two of degree 3 (which we might have concluded immediately from the fact that the derived quotient of G has order 9).

84.4 Operations on Group Algebras and their Subalgebras

84.4.1 Operations on Group Algebras

The operations in this section can only be applied to a full group algebra. Functions accepting also a subalgebra of type AlgGrpSub are dealt with in the next section.

```
Algebra(A)
```

For a group algebra A given in vector representation, construct the associative structure constant algebra B isomorphic to A together with the isomorphism $A \to B$.

```
AugmentationMap(A)
```

The augmentation map of A. That is, the map $A \to R : \sum_{g\in G} r_g * g \to \sum_{g\in G} r_g$.

```
AugmentationIdeal(A)
```

The augmentation ideal of the group algebra A given in vector representation. This is defined as the kernel of the augmentation map.

```
RepresentationType(A)
```

Given a group algebra A, return either `"Vector"` or `"Terms"` depending on which representation is used for the elements of A.

```
ChangeRepresentationType(A, Rep)
```

Given a group algebra A, construct an isomorphic group algebra B in which the elements are represented as specified by `Rep` which may be `"Vector"` or `"Terms"`, together with the homomorphism from A to B.

```
ConstructTable(A)
```

Procedure which, given a group algebra $A = R[G]$ in vector representation, constructs the multiplication table for the group G to speed up multiplication in A. If the multiplication table already exists, nothing is done.

```
CoefficientRing(A)
BaseRing(A)
```

The coefficient ring (base ring) of A.

84.4.2 Operations on Subalgebras of Group Algebras

The functions in this section can be applied to group algebras and their subalgebras of type `AlgGrpSub`.

```
S ! 1
```

Create the identity element of the group algebra (subalgebra) S. Note that for a proper subalgebra of the full group algebra this may be different from the identity element of the group.

```
Group(S)
```

The group G for the group algebra (subalgebra) S.

```
GroupAlgebra(S)
```

The group algebra of which S is a subalgebra.

```
Module(S)
```

For a subalgebra S of the group algebra $A = R[G]$, return the submodule of the module underlying A which corresponds to S. This is an R-module of dimension `Dimension(S)` and degree `Dimension(A)`. Also returns (as a second return value) the natural map from the subalgebra to the module.

```
CoefficientRing(A)
BaseRing(A)
```

The coefficient ring (base ring) of A.

```
BasisMatrix(S)
```

For a subalgebra S of the group algebra $A = R[G]$ return the coefficient matrix of the basis of S with respect to the basis of A. If S has dimension m this is an $m \times |G|$-matrix over R where the i-th row are the coefficients of the i-th basis vector of S with respect to the fixed basis of A.

```
Coordinates(S, a)
```

Given an element a which lies in the subalgebra S, return a sequence giving the coordinates of a with respect to the basis of S.

```
IsLeftIdeal(S)
```

Returns `true` if S is a left ideal of its group algebra; otherwise `false`.

```
IsRightIdeal(S)
```

Returns `true` if S is a right ideal of its group algebra; otherwise `false`.

```
IsIdeal(S)
```

Returns `true` if S is a (two-sided) ideal of its group algebra; otherwise `false`.

```
Centraliser(S)
Centralizer(S)
```

The centralizer of the subalgebra S of a group algebra A (in A).

```
Idealiser(S)
Idealizer(S)
```

The largest subalgebra T of A such that S is an ideal in T.

```
LeftAnnihilator(S)
```

For a subalgebra S of the group algebra A construct the left annihilator of S, that is, the subalgebra of A consisting of all elements a such that $a * s = 0$ for all $s \in S$.

```
RightAnnihilator(S)
```

For a subalgebra S of the group algebra A construct the right annihilator of S, that is, the subalgebra of A consisting of all elements a such that $s * a = 0$ for all $s \in S$.

Example H84E4

We construct the group algebra of an elementary abelian group over $GF(2)$ and get its Jacobson ideal.

```
> A := AbelianGroup([2,2,2,2,2]);
> FG := GroupAlgebra(GF(2), A);
> J := JacobsonRadical(FG);
> J;
Ideal of dimension 31 of the group algebra FG
```

We now check that the Jacobson radical is nilpotent and get its nilpotency class.

```
> JPow := [ J ];
> I := J;
> while Dimension(I) ne 0 do
>     I := I*J;
>     Append(~JPow, I);
> end while;
> [ Dimension(I) : I in JPow ];
[ 31, 26, 16, 6, 1, 0 ]
```

Thus, J is nilpotent of class 6. However, every non-zero element of J is of course nilpotent of class 2.

```
> IsNilpotent(Random(J));
true 2
```

84.5 Operations on Elements

The operations in this section can be applied to elements of either a group algebra or of a group algebra subalgebra of type `AlgGrpSub`. Only those operations are listed, which are additional to those available for general algebras.

`a + r`

`r + a`

The sum of the group algebra element $a \in R[G]$ and the scalar $r \in R$.

`a + g`

`g + a`

The sum of the group algebra element $a \in R[G]$ and the group element $g \in G$.

`a - r`

`r - a`

The difference of the group algebra element $a \in R[G]$ and the scalar $r \in R$.

`a - g`

`g - a`

The difference of the group algebra element $a \in R[G]$ and the group element $g \in G$.

`a * r`

`r * a`

The product of the group algebra element $a \in R[G]$ and the scalar $r \in R$.

`g * a`

`a * g`

The product of the group algebra element $a \in R[G]$ and the group element $g \in G$.

`Support(a)`

The support of a; that is, the sequence of group elements whose coefficients in a are non-zero.

`Trace(a)`

The trace of a; that is, the coefficient of 1_G in a.

`Augmentation(a)`

The augmentation of the group algebra element a; that is, $\sum_{g \in G} r_g$ where $a = \sum_{g \in G} r_g * g$.

`Involution(a)`

If $a = \sum_{g \in G} r_g * g$, returns $\sum_{g \in G} r_g * g^{-1}$.

`Coefficient(a, g)`

`a[g]`

The coefficient of $g \in G$ in $a \in R[G]$.

`ElementToSequence(a)`

`Eltseq(a)`

If a is an element from a group algebra A given in vector representation, this returns the sequence of coefficients with respect to the fixed basis of A. If A is given in terms representation, this returns a sequence of tuples, where the second entry is a group element and the first is the coefficient of that group element in a.

`Coefficients(a)`

For an element a from a group algebra A given in vector representation, this returns the sequence of coefficients with respect to the fixed basis of A.

```
Centraliser(a)
Centralizer(a)
```

The centralizer in the group algebra A of the element a of A.

```
Centraliser(S, a)
Centralizer(S, a)
```

The centralizer of the element a (of a group algebra A) in the subalgebra S of A.

Example H84E5

We use the group algebra to determine the diameter of the Cayley graph of a group.

```
> G := Alt(6);
> QG := GroupAlgebra( Rationals(), G );
> e := QG!1 + &+[ QG!g : g in Generators(G) ];
> e;
Id(G) + (1, 2)(3, 4, 5, 6) + (1, 2, 3)
```

The group elements that can be expressed as words of length at most n in the generators of G have non-zero coefficient in e^n. The following function returns for a group algebra element e a sequence with the cardinalities of the supports of e^n and breaks when the group order is reached.

```
> wordcount := function(e)
>     f := e;
>     count := [ #Support(f) ];
>     while count[#count] lt #Group(Parent(e)) do
>         f *:= e;
>         Append(~count, #Support(f));
>     end while;
>     return count;
> end function;
```

Now apply this function to the above defined element:

```
> wordcount( e );
[ 3, 7, 14, 26, 47, 83, 140, 219, 293, 345, 360 ]
```

Thus, every element in A_6 can be expressed as a word of length at most 11 in the generators $(1,2)(3,4,5,6)$ and $(1,2,3)$. A better 2-generator set is for example $(1,2,3,4,5)$ and $(1,5,3,6,4)$, where all elements can be expressed as words of length at most 10 and this is in fact optimal. A worst 2-generator set is given by $(1,2)(3,4)$ and $(1,5,3,2)(4,6)$.

```
> wordcount( QG!1 + G!(1,2,3,4,5) + G!(1,5,3,6,4) );
[ 3, 7, 15, 31, 60, 109, 183, 274, 350, 360 ]
> wordcount( QG!1 + G!(1,2)(3,4) + G!(1,5,3,2)(4,6) );
[ 3, 6, 11, 18, 28, 43, 63, 88, 119, 158, 206, 255, 297, 329, 352, 360 ]
```

Example H84E6

The group algebra can also be used to investigate the random distribution of words of a certain length in the generators of the group.

```
> M11 := sub< Sym(11) | (1,11,9,10,4,3,7,2,6,5,8), (1,5,6,3,4,2,7,11,9,10,8) >;
> A := GroupAlgebra(RealField(16), M11 : Rep := "Vector");
> A;
Group algebra with vector representation
Coefficient ring: Real Field of precision 16
Group: Permutation group M11 acting on a set of cardinality 11
    Order = 7920 = 2^4 * 3^2 * 5 * 11
        (1, 11, 9, 10, 4, 3, 7, 2, 6, 5, 8)
        (1, 5, 6, 3, 4, 2, 7, 11, 9, 10, 8)
> e := (A!M11.1 + A!M11.2) / 2.0;
> eta := Eta(A) / #M11;
```

For growing n, the words of length n in the generators of `M11` converge towards a random distribution iff `e`^{n} converges towards `eta`. We look at the quadratic differences of the coefficients of `e`^{n}`-eta` for $n = 10, 20, 30, 40, 50$.

```
> e10 := e^10;
> f := A!1;
> for i in [1..5] do
>     f *:= e10;
>     print &+[ c^2 : c in Eltseq(f - eta) ];
> end for;
0.0012050667195213
1.289719354694155e-5
5.9390965208879e-7
3.394099291966e-8
2.19432454574986e-9
```

85 BASIC ALGEBRAS

Chapter 85
BASIC ALGEBRAS

85.1 Introduction

A basic algebra is a finite dimensional algebra A over a field, all of whose simple modules have dimension one. In the literature such an algebra is known as a “split” basic algebra. Every algebra is Morita equivalent to a basic algebra, though a field extension may be necessary to obtain the split basic algebra. Magma has several functions that create the basic algebras corresponding to algebras of different types.

The type `AlgBas` in Magma is optimized for the purposes of doing homological calculations. A basic algebra A is generated by elements $a_1, a_2, \ldots, a_t$ where $a_1, \ldots, a_s$ are the primitive idempotent generators and $a_{s+1}, \ldots, a_t$ are the nonidempotent generators. Each nonidempotent generator, a_k must have the property that $a_i * a_k * a_j = a_k$ for specific idempotent generators a_i and a_j. The projective indecomposable modules have the form $P_i = a_i \cdot A$ for $i = 1, \ldots, s$ and the simple modules have the form $S_i = P_i/\mathrm{Rad}(P_i)$, where $\mathrm{Rad}(P_i)$ is the radical of P_i.

85.2 Basic Algebras

In the Magma implementation the algebra is given as the sequence of projective modules $P_1, \ldots, P_s$ together with a path tree for each projective module. A projective module consists of a matrix for each generator $a_1, a_2, \ldots, a_t$ giving the action of the generator on the vector space of the module. The basis $b_1, b_2, \ldots, b_n$ for the vector space of P_i is chosen so that each basis element is the image of a basis element of lower index under multiplication by a nonidempotent generator of A. The structure of the basis is recorded in the path tree which is a sequence $[< 1, i >, < j, k >, \ldots]$ of 2-tuples of length $n =$ `Dimension`(P_i). The first entry $< 1, i >$ indicates that $b_1 = b_1 * a_i$ where a_i is the primitive idempotent in the algebra A such that $P_i = A \cdot a_i$. Similarly, if entry number k in the path tree is $\langle u, v\rangle$ then $b_k = b_u * a_v$ where $v > s$ if $k > 1$.

85.2.1 Creation

The first function for creating a basic algebra is the most basic, in which the user supplies the projective modules and the path trees directly.

```
BasicAlgebra(Q)
```

Given a sequence $[Q_i, \ldots, Q_s]$ of 2-tuples such that each $Q_i =< M_i, T_i >$ consisting of a module for a matrix algebra M_i and a path tree T_i for M_i, the function creates the basic algebra whose projective modules are the first entries $M_1, \ldots, M_s$ and the path trees are the corresponding second entries.

The next two functions create a basic algebra from generators and relations. The user must, additionally, specify the quiver with relations, giving the number of idempotents and the beginning and end points of the arrows in the quiver.

```
BasicAlgebra(F,R,s,P)
```

Creates the basic algebra given by the presentation. Here F is a free algebra and R is the sequence of relation for the nonidempotent generators of the algebra. If the free algebra F is generated by elements $a_1, \ldots, a_t$, the function assumes that $a_1, \ldots, a_s$ are the mutually orthogonal primitive idempotents and it creates all of the appropriate relations including $a_1 + \ldots + a_s = 1$. The nonidempotent generators are then $a_{s+1}, \ldots, a_t$. So $\ell_k = \langle i, j\rangle$ for $i, j \leq s$ means that $a_{s+k} = a_i * a_{s+k} * a_j$. Each of the relations in R is given as a linear combination of words in the nonidempotent generators $a_{s+1}, \ldots, a_t \in F$. The sequence P is a sequence of 2-tuples, one for each nonidempotent generator, giving the beginning and ending nodes of the generator. That is, each tuple is the pair of indices of the idempotents which multiply as the identity on the nonidempotent generator on the left and on the right.

```
BasicAlgebra(F,R)
```

Creates the basic algebra of a local algebra from the presentation of the algebra. Here F is a free algebra whose variable represent the nonidempotent generators and R is the sequence of relations among those variables.

```
TensorProduct(A, B)
```

The tensor product of the basic algebras A and B.

The modular group algebra of a p-group is naturally a basic algebra. The next function create the basic algebra from the group information.

```
BasicAlgebra(G, k)
```

```
BasicAlgebra(G, k)
```

```
BasicAlgebra(G)
```

Given a finite p-group G and a finite field k of characteristic p, returns the group algebra kG in the form of a basic algebra. If no field k is supplied then the prime field of characteristic p is assumed to be the field of coefficients.

85.2.2 Special Basic Algebras

There are several functions that create basic algebras of special interest. The most basic of these creates the basic algebra that is Morita equivalent to a matrix algebra defined over a finite field by first condensing the algebra and then splitting the irreducible modules as needed.

Included among these constructions are the basic algebras of Schur algebras and Hecke algebras over finite fields. The Schur algebras arise in the representation theory of symmetric groups. The Schur algebras have finite global dimension and hence their ext-algebras have finite dimension. By a Hecke algebra, we mean the algebra of endomorphisms of a permutation module of a finite group.

```
BasicAlgebra(A)
```

```
BasicAlgebraOfMatrixAlgebra(A)
```

This function creates the split basic algebra of the matrix algebra A. The function first produces a presentation and condensed algebra for A. In the event that the field of coefficients k of A is not a splitting field, then the returned basic algebra is defined over the minimal extension of k that is a splitting field for A.

```
BasicAlgebraOfEndomorphismAlgebra(M)
```

Returns the split basic algebra of the endomorphism ring of the module M. In the event that the field of coefficients of M is not a splitting field for M, then the field is extended and the basic algebra is defined over the minimal extension needed to split M.

```
BasicAlgebraOfHeckeAlgebra(G, H, F)
```

Returns the basic algebra of the Hecke algebra which is the algebra of endomorphisms of the permutation module over G with point stabilizer H and field of coefficients F. In the event that F is not a splitting field then the field F is extended to a splitting field which is the field of coefficients of the returned basic algebra.

```
BasicAlgebraOfSchurAlgebra(n, r, F)
```

Creates the basic algebra of the Schur algebra $S(n,r)$ over the field F. The Schur algebra is the algebra of endomorphisms of the module over the symmetric group $Sym(r)$ that is the tensor product of r copies of a vector space V over F of dimension n, the group $Sym(r)$ acting by permuting the r copies.

```
BasicAlgebraOfGroupAlgebra(G,F)
```

```
BasicAlgebraOfGroupAlgebra(G,F)
```

```
BasicAlgebraOfGroupAlgebra(G,F)
```

The function returns the basic algebra of the group algebra of G with coefficients in the field F. The function requires the creation of the projective indecomposable FG-modules. In the event that the field F is not a splitting field for the irreducible FG-modules then the base ring of the returned algebra is the minimal extension of F that is necessary to get a splitting field.

```
BasicAlgebra(S)
```

Returns the basic algebra of the action algebra on the module which is the direct sum of the modules in the sequence S. In the event that the irreducible composition factors of the modules in S are not absolutely irreducible, then the returned basic algebra is defined over the splitting field for the irreducible modules.

```
BasicAlgebraOfBlockAlgebra(S)
```

Returns the basic algebra of the block algebra, the projective modules of which are given in the sequence S. In the event that the irreducible composition factors of the modules in S are not absolutely irreducible, then the returned basic algebra is defined over the splitting field for the irreducible modules.

```
BasicAlgebraOfPrincipalBlock(G,k)
```

Returns the basic algebra of the principal block of the group algebra of G. If the simple modules in the principal kG block are all absolutely simple, then the ordering of the projective modules for the returned basic algebra is exactly the same as the ordering of the projectives modules in the principal block returned by the function `IndecomposableProjectives`. Otherwise, the base ring of the returned basic algebra is the least extension of k necessary to split the simple modules in the principal block and the simple modules of the returned algebra are ordered by increasing dimension of the corresponding simple modules of kG.

```
BasicAlgebraOfExtAlgebra(A)
```

The function returns the basic algebra from a computed ext-algebra which is $\mathrm{Ext}^*_A(S,S)$, where S is the direct sum of the irreducible A-modules, of the basic algebra A. If no ext-algebra for A has been computed or if the exalgebra is not verified to be finite dimensional then an error is returned.

```
BasicAlgebraOfExtAlgebra(A)
```

The function returns the basic algebra for the ext-algebra, which is $\mathrm{Ext}^*_A(S,S)$, where S is the direct sum of the irreducible A-modules, of A computed to n steps. If no ext-algebra for A to n steps has been computed then it will be computed. If the ext-algebra is not verified to be finite dimensional by the computation, then an error is returned.

```
BasicAlgebraOfExtAlgebra(A)
```

The function creates the basic algebra from a computed ext-algebra. The input *ext* is the output of the `ExtAlgebra` function. If the ext-algebra is not verified to be finite dimensional by the computation, then an error is returned.

```
OppositeAlgebra(B)
```

The opposite algebra of the basic algebra B. The opposite algebra of B is the algebra with the same set of elements and the same addition but with multiplication $*$ given by $a * b = ba$ for a and b in A.

Example H85E1

We form the basic algebra of the group algebra of the alternating group on 6 letters in characteristic 2.

```
> G := AlternatingGroup(6);
> A := BasicAlgebraOfGroupAlgebra(G, GF(2));
> A;
Basic algebra of dimension 36 over GF(2^2)
Number of projective modules: 5
Number of generators: 9
```

Note that the field of coefficients has been extended in order to split the irreducible G-modules. A great deal of information on the nature of the group algebra and its basic algebra can be obtained from analysis such as the following.

```
> [Dimension(ProjectiveModule(A,i)): i in [1 .. 5]];
[ 9, 9, 16, 1, 1 ]
> prj := CompactProjectiveResolutionsOfAllSimpleModules(A,8);
> [x`BettiNumbers:x in prj];
[
    [
        [ 1, 0, 0, 0, 0 ],
        [ 0, 0, 1, 0, 0 ],
        [ 1, 0, 0, 0, 0 ],
        [ 1, 0, 0, 0, 0 ],
        [ 0, 0, 1, 0, 0 ],
        [ 1, 0, 0, 0, 0 ],
        [ 1, 0, 0, 0, 0 ],
        [ 0, 0, 1, 0, 0 ],
        [ 1, 0, 0, 0, 0 ]
    ],
    [
        [ 0, 1, 0, 0, 0 ],
        [ 0, 0, 1, 0, 0 ],
        [ 0, 1, 0, 0, 0 ],
        [ 0, 1, 0, 0, 0 ],
        [ 0, 0, 1, 0, 0 ],
        [ 0, 1, 0, 0, 0 ],
        [ 0, 1, 0, 0, 0 ],
        [ 0, 0, 1, 0, 0 ],
        [ 0, 1, 0, 0, 0 ]
    ],
    [
        [ 0, 0, 3, 0, 0 ],
        [ 1, 1, 2, 0, 0 ],
        [ 0, 0, 3, 0, 0 ],
        [ 0, 0, 2, 0, 0 ],
        [ 1, 1, 1, 0, 0 ],
        [ 0, 0, 2, 0, 0 ],
        [ 0, 0, 1, 0, 0 ],
        [ 1, 1, 0, 0, 0 ],
        [ 0, 0, 1, 0, 0 ]
    ],
    [
        [ 0, 0, 0, 0, 0 ],
        [ 0, 0, 0, 0, 0 ],
        [ 0, 0, 0, 0, 0 ],
        [ 0, 0, 0, 0, 0 ],
        [ 0, 0, 0, 0, 0 ],
        [ 0, 0, 0, 0, 0 ],
        [ 0, 0, 0, 0, 0 ],
        [ 0, 0, 0, 0, 0 ],
        [ 0, 0, 0, 1, 0 ]
    ],
    [
        [ 0, 0, 0, 0, 0 ],
        [ 0, 0, 0, 0, 0 ],
        [ 0, 0, 0, 0, 0 ],
        [ 0, 0, 0, 0, 0 ],
        [ 0, 0, 0, 0, 0 ],
        [ 0, 0, 0, 0, 0 ],
        [ 0, 0, 0, 0, 0 ],
        [ 0, 0, 0, 0, 0 ],
        [ 0, 0, 0, 0, 1 ]
    ]
]
```

So we see that the last two simple modules are projective and hence in the group algebra FG, they represent blocks of defect 0. From the information on the Betti numbers we observe that the first two simple modules have periodic projective resolutions. Thus the third simple module for the basic algebra corresponds to the trivial FG-module, which we know is neither projective nor periodic.

Example H85E2

We construct the Schur algebra $S(3, 7)$ with coefficients in a field of characteristic 2.

```
> A := BasicAlgebraOfSchurAlgebra(3,6,GF(2));
> A;
Basic algebra of dimension 58 over GF(2)
Number of projective modules: 7
Number of generators: 21
```

It is known that A has finite global dimension, hence its ext-algebra has finite dimension.

```
> B := BasicAlgebraOfExtAlgebra(A,10);
> B;
Basic algebra of dimension 56 over GF(2)
Number of projective modules: 7
Number of generators: 25
```

We check to see if the ext-algebra of B might have finite dimension.

```
> SumOfBettiNumbersOfSimpleModules(B,5);
600
> SumOfBettiNumbersOfSimpleModules(B,6);
1334
> SumOfBettiNumbersOfSimpleModules(B,7);
3008
```

The sum of the Betti number would be the dimension of the ext-algebra computed to the indicated degree. It would appear that the ext-algebra is infinite dimensional. Just to check, we look at a particular simple module, chosen randomly.

```
> CompactProjectiveResolution(SimpleModule(B,4),10)‘BettiNumbers;
[
    [ 8, 0, 81, 3, 61, 69, 55 ],
    [ 3, 0, 36, 1, 26, 30, 24 ],
    [ 1, 0, 16, 1, 11, 13, 10 ],
    [ 1, 0, 7, 1, 5, 5, 4 ],
    [ 1, 0, 3, 1, 2, 2, 1 ],
    [ 1, 0, 1, 0, 1, 1, 0 ],
    [ 0, 0, 0, 0, 1, 1, 0 ],
    [ 0, 0, 0, 0, 1, 0, 1 ],
    [ 0, 0, 0, 1, 0, 1, 0 ],
    [ 1, 0, 0, 0, 0, 0, 1 ],
    [ 0, 0, 0, 1, 0, 0, 0 ]
]
```

Thus we have strong evidence that the fourth simple module has infinite projective dimension. In that case, the ext-algebra of B is not finite dimensional.
Now we consider the same example except in characteristic 3.

```
> A := BasicAlgebraOfSchurAlgebra(3,6,GF(3));
> A;
Basic algebra of dimension 48 over GF(3)
Number of projective modules: 7
Number of generators: 21
> B := BasicAlgebraOfExtAlgebra(A,10);
> B;
Basic algebra of dimension 98 over GF(3)
Number of projective modules: 7
Number of generators: 21
> SumOfBettiNumbersOfSimpleModules(B,5);
48
> SumOfBettiNumbersOfSimpleModules(B,6);
48
```

So we see that the algebra B has global dimension at most 5. We can compute its ext-algebra.

```
> C := BasicAlgebraOfExtAlgebra(B,10);
> C;
Basic algebra of dimension 48 over GF(3)
Number of projective modules: 7
Number of generators: 21
> D := BasicAlgebraOfExtAlgebra(C,10);
> D;
Basic algebra of dimension 98 over GF(3)
Number of projective modules: 7
Number of generators: 21
```

This provides evidence that A is isomorphic to its double ext-algebra. This would suggest that A might be a Koszul algebra.

85.2.3 Access Functions

These functions return basic information, underlying structures and elements of the given basic algebras.

`B . i`

The i^{th} element in the standard basis for the underlying vector space of the algebra B.

`BaseRing(B)`

`CoefficientRing(B)`

Given an algebra B over a field k the function returns k.

`VectorSpace(B)`

`KSpace(B)`

The underlying k-vector space of the algebra B. The space is the direct sum of the underlying vector space of the indecomposable projective modules.

`Dimension(B)`

The dimension of the underlying vector space of the algebra B.

`Basis(B)`

A basis of the underlying vector space of the algebra B.

`Generators(B)`

The generators of the algebra B as a sequence of elements in the underlying vector space of the algebra B.

`IdempotentGenerators(B)`

The sequence of mutually orthogonal idempotent generators of the basic algebra B as elements in the underlying vector space of B.

```
IdempotentPositions(B)
```

The sequence $N = [n_1, \ldots, n_s]$ such that $B.n_1, \ldots, B.n_s$ are the mutually orthogonal idempotent generators of the algebra B.

```
NonIdempotentGenerators(B)
```

The sequence of nonidempotent generators of the basic algebra B as elements in the underlying vector space of the algebra B.

```
Random(B)
```

A random element of the algebra B as an element of the underlying vector space of B.

```
NumberOfProjectives(B)
```

The number of nonisomorphic indecomposable projective modules in the basic algebra B.

```
NumberOfGenerators(B)
Ngens(B)
```

The number of generators (idempotent and nonidempotent) of the basic algebra B.

```
DimensionsOfProjectiveModules(B)
```

The sequence of the dimensions of the projective modules of the basic algebra B.

```
DimensionsOfInjectiveModules(B)
```

The sequence of the dimensions of the injective modules of the basic algebra B.

85.2.4 Elementary Operations

```
a + b
```

The sum of the two elements a and b.

```
a * b
```

The product of the two elements a and b.

```
a ^ n
```

The n^{th} power of the element a.

Example H85E3

We create the basic algebra of the quiver with three nodes and three arrows over the field with 7 elements. The first arrow (a) goes from node 1 to node 2, the second (b) from node 2 to node 1, and (c) from node 2 to node 3. The arrows satisfy the relation that $(a * b)^3 = 0$.

```
> ff := GF(7);
> FA<e1,e2,e3,a,b,c> := FreeAlgebra(ff,6);
> rrr := [a*b*a*b*a*b];
> D := BasicAlgebra(FA,rrr,3,[<1,2>,<2,1>,<2,3>]);
> D;
Basic algebra of dimension 21 over GF(7)
Number of projective modules: 3
Number of generators: 6
> DimensionsOfProjectiveModules(D);
[ 9, 11, 1 ]
> DimensionsOfInjectiveModules(D);
[ 6, 7, 8 ]
```

So we can see that the algebra is not self-injective.
Now we can check the nilpotence degree of the radical of D. The radical of D is generated by the nonidempotent generators.

```
> S := NonIdempotentGenerators(D);
> S;
[ (0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0), (0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0
0 0 0 0 0), (0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0) ]
> S2 := [x*y:x in S, y in S|x*y ne 0];
> #S2;
3
> S3 := [x*y:x in S2, y in S|x*y ne 0];
> #S3;
3
> S4 := [x*y:x in S3, y in S|x*y ne 0];
> #S4;
3
> S5 := [x*y:x in S4, y in S|x*y ne 0];
> #S5;
3
> S6 := [x*y:x in S5, y in S|x*y ne 0];
> #S6;
2
> S7 := [x*y:x in S6, y in S|x*y ne 0];
> #S7;
1
> S8 := [x*y:x in S7, y in S|x*y ne 0];
> #S8;
0
```

Example H85E4

First we create the basic algebra for the symmetric group S_3 over the field $GF(3)$

```
> FA<e1,e2,a,b> := FreeAlgebra(GF(3),4);
> MM:= [a*b*a, b*a*b];
> BS3 := BasicAlgebra(FA, MM, 2, [<1,2>,<2,1>]);
> BS3;
Basic algebra of dimension 6 over GF(3)
Number of projective modules: 2
Number of generators: 4
> DimensionsOfProjectiveModules(BS3);
[ 3, 3 ]
```

Next we create the basic algebra for the cyclic group of order 3.

```
> gg := CyclicGroup(3);
> BC3 := BasicAlgebra(gg,GF(3));
> BC3;
Basic algebra of dimension 3 over GF(3)
Number of projective modules: 1
Number of generators: 2
```

We create the basic algebra for the direct product $C_3 \times S_3$.

```
> A := TensorProduct(BS3,BC3);
> A;
Basic algebra of dimension 18 over GF(3)
Number of projective modules: 2
Number of generators: 6
> DimensionsOfProjectiveModules(A);
[ 9, 9 ]
```

Example H85E5

We create the basic algebra for A_4 over a field with 2 elements. The group algebra has two nonisomorphic projective modules. We define the basic algebra by constructing the matrix algebra for the projective modules and the path trees and entering this data into the `BasicAlgebra` function.

Note that the matrices are sparse so we will define them by specifying the nonzero rows.

```
> ff := GF(2);
> MM6 := MatrixAlgebra(ff,6);
> e11 := MM6!0;
> e12 := MM6!0;
> VV6 := VectorSpace(GF(2),6);
> BB6 := Basis(VV6);
> e11[1]  := BB6[1];
> e11[3]  := BB6[3];
> e11[4]  := BB6[4];
> e11[6]  := BB6[6];
> e12[2]  := BB6[2];
> e12[5]  := BB6[5];
> a1 := MM6!0;
> b1 := MM6!0;
> c1 := MM6!0;
> d1 := MM6!0;
> a1[1] := BB6[2];
> b1[1] := BB6[3];
> c1[2] := BB6[4];
> a1[3] := BB6[5];
> b1[4] := BB6[6];
> c1[5] := BB6[6];
> A1 := sub< MM6 | [e11, e12, a1, b1, c1, d1] >;
> T1 := [ <1,1>, <1,3>, <1,4>, <2,5>, <3,3>, <4,4>];
>
> VV5 := VectorSpace(ff,5);
> BB5 := Basis(VV5);
> MM5 := MatrixAlgebra(ff,5);
> e21 := MM5!0;
> e22 := MM5!0;
> e22[1]  := BB5[1];
> e22[3]  := BB5[3];
> e22[5]  := BB5[5];
> e21[2]  := BB5[2];
> e21[4]  := BB5[4];
> a2 := MM5!0;
> b2 := MM5!0;
> c2 := MM5!0;
> d2 := MM5!0;
> f2 := MM5!0;
> g2 := MM5!0;
> c2[1] := BB5[2];
> d2[1] := BB5[3];
> b2[2] := BB5[4];
> d2[3] := BB5[5];
> a2[4] := BB5[5];
> A2 := sub< MM5 | [e21, e22, a2, b2, c2, d2] >;
> T2 := [<1,2>, <1,5>, <1,6>, <2,4>, <3,6>];
>
> C := BasicAlgebra( [<A1, T1>, <A2, T2>] );
> C;
Basic algebra of dimension 11 over GF(2)
Number of projective modules: 2
Number of generators: 6
> DimensionsOfProjectiveModules(C);
[ 6, 5 ]
> DimensionsOfInjectiveModules(C);
```

```
[ 6, 5 ]
```

85.2.5 Boolean Functions

A basic algebra in Magma is a sequence of matrix algebras, each with a path tree. When a basic algebra is created by entering such a sequence, Magma does not check to see if all of the properties of a basic algebra are satisfied, as this is an expensive operation. The following two function check to see if the properties of a basic algebra are satisfied.

`IsDimensionCompatible(B)`

Returns `true` if the dimension of a basic algebra is the same as the dimension of the matrix algebra of its action on itself. If false then the algebra is not a basic algebra.

`IsPathTree(B)`

Returns `true` if the basis elements of the projective modules in the basic algebra are determined by the path tree. If false, then the algebra is not a true basic algebra.

Two other functions check commutativity.

`IsCommutative(A)`

Returns `true` if the basic algebra A is commutative.

`IsCentral(A,x)`

Returns `true` if the element x is in the center of the basic algebra A.

85.3 Homomorphisms

Magma has the capability of creating homomorphisms of basic algebras. A homomorphism of a basic algebra has the type Map. The matrix of the homomorphism ϕ is accessed by entering `Matrix(phi)`. The kernel of a homomorphism is returned as a subspace of the vector space of the domain of the algebra. This is a two-sided ideal. There is no special type for ideals. They are subspaces of the underlying vector space of the algebra. They can be generated from any given set of elements of the algebra

The image of a homomorphism is returned as a basic algebra, together with the embedding homomorphism.

A homomorphism from a basic algebra A to a basic algebra B is normally created from a matrix having dimension of A rows and dimension of B columns. Note that Magma does not automatically check to see if the created map is an algebra homomorphism.

`hom< A - >`

The algebra homomorphism from basic algebra A to basic algebra B, whose matrix is the matrix S. Given a map ϕ, a homomorphism of basic algebra, the matrix of that map is recalled with the command `Matrix(phi)`.

```
Kernel(phi)
```

The kernel of the map ϕ.

```
Image(phi)
```

The image of the homomorphism ϕ together with the embedding homomorphism of the image into the codomain of ϕ.

```
IsAlgebraHomomorphism(A, B, psi)
```

Return `true` if the matrix ψ represents a homomorphism from basic algebra A to basic algebra B.

```
X * Y
```

The composition of the maps X and Y.

```
IsAlgebraHomomorphism(A, B, psi)
IsAlgebraHomomorphism(A, B, psi)
IsAlgebraHomomorphism(A, B, psi)
IsAlgebraHomomorphism(A, B, psi)
```

Returns `true`, if the map ψ is a homomorphism of basic algebras

```
IsAlgebraHomomorphism(psi)
```

Returns `true` if the map ψ is a homomorphism of basic algebras.

85.4 Subalgebras and Quotient Algebras

A subalgebra is also returned with the embedding homomorphism, and a quotient algebra is returned with the natural quotient map. These are needed for creating some standard subalgebras such as the centre of the algebra.

85.4.1 Subalgebras and their Constructions

```
sub< A | S >
```

The subalgebra of A generated by the elements of the sequence S, together with the inclusion map of the subalgebra into A. The subalgebra contains the idempotent of minimal rank in A that acts as a multiplicative identity on the elements of S.

```
SubalgebraFromBasis(A, V)
```

Given a basic algebra A and the basis V of a subspace of A, the function returns the basic algebra which is the subalgebra spanned by the subspace and the inclusion matrix of the homomorphism embedding the subalgebra into A. Note that the space V might not contain the identity element of V and in that case the minimal possible identity element is added to the returned subalgebra.

```
MaximalIdempotent(A, S)
```

Given a basic algebra A, a subspace S of the vector space of A that is closed under multiplication, this function returns an idempotent in A which has maximal rank among all idempotents contained in S.

```
MinimalIdentity(A, S)
```

Returns the idempotent of smallest rank that is a two sided identity for the elements in the set S.

```
Centre(A)
```

The centre of the basic algebra as a basic algebra together with the inclusion homomorphism.

```
Centralizer(A,S)
```

Returns the centralizer in the basic algebra A of the elements in the sequence S, along with the homomorphism embedding the centralizer into A.

```
MaximalCommutativeSubalgebra(A,S)
```

Returns a maximal commutative subalgebra of the basic algebra A that contains the elements of the sequence S. An error occurs if the elements of S do not commute.

85.4.2 Ideals and their Construction

```
ideal<  A | S >
```

```
ideal<  A | S >
```

Returns the subspace of the vector space of the algebra A that is the ideal of the A generated by the given sequence of elements S.

```
LeftAnnihilator(A, S)
```

Returns a basis for the left annihilator of the sequence S of elements in the basic algebra A.

```
RightAnnihilator(A, S)
```

Returns a basis for the right annihilator of the sequence S of elements in the basic algebra A.

```
Annihilator(A,S)
```

Returns a basis for the two-sided annihilator of the sequence of elements S of the basic algebra A.

```
IsIdeal(A, S)
```

Returns `true` if the subspace spanned by the elements of S is a two-sided ideal of the basic algebra A.

```
IsLeftIdeal(A,S)
```

Returns `true` if the subspace spanned by the elements of S is a left ideal of the basic algebra A.

```
IsRightIdeal(A, S)
```

Returns `true` if the subspace spanned by the elements of S is a two-sided ideal of the basic algebra A.

```
RandomIdealGeneratedBy(A, n)
```

Returns the ideal generated by n randomly selected elements in the Jacobson radical of the basic algebra A.

85.4.3 Quotient Algebras

```
quo<  A | S >
```

Returns the quotient algebra of A by the ideal S, which is a subspace of the vector space of A, together with the quotient map.

```
CoverAlgebra(A)
```

Constructs the maximal extension B as in $[0 \to K \to B \to A \to 0]$ such that B acts trivially on K and B is an algebra with exactly the same minimal number of generators as A. Returns B and the algebra homomorphism of B onto A.

```
GradedCoverAlgebra(A)
```

This assumes that we are given the truncated algebra of a graded algebra. It creates the basic algebra of the natural cover of A and also returns the matrix of the cover onto A.

```
TruncatedAlgebra(A,n)
```

The quotient of the algebra by the n^{th} power of the radical of A. Returns also the quotient map.

85.5 Minimal Forms and Gradings

There are some standard ways of rewriting an algebra to an isomorphic form. We can also constructed the associated graded algebra of a basic algebra. Also included here is a function for rearranging the orders of idempotents in a basic algebra.

The first intrinsic is used extensively in the programs for computing automorphisms and isomorphisms.

```
MinimalGeneratorForm(A)
```

Returns a record consisting of an isomorphic basic algebra having the property that it is generated by a minimal number of elements and the projective modules are filtered by radical layers. The record consists of the following fields:

(a) The algebra in minimal form (field `algebra`).

(b) The map from the minimal generator form algebra to A (field `Homomorphism`).

(c) The inverse map from A to the minimal generator form algebra (field `InverseHomomorphism`).

(d) The dimensions of the radical layer (field `RadicalDimensions`).

(e) The dimensions of the filtration of the top radical layer by the socle layer (field `FilterDimensions`).

```
MinimalGeneratorFormAlgebra(A)
```

Returns an isomorphic algebra having minimal generator form.

```
AssociatedGradedAlgebra(A)
```

Returns the basic algebra that is isomorphic to the associated graded algebra of A.

```
GradedCapHomomorphism(A)
```

Returns the matrix of the map from $A/Rad(A)$ to $X/Rad(X)$ where X is the associated graded algebra of A.

```
GradedCapHomomorphism(A, B, mu)
```

Given an algebra homomorphism $mu : A \rightarrow B$, returns the induced homomorphism $A/\operatorname{Rad}^2(A) \rightarrow B/\operatorname{Rad}^2(B)$, where Rad^2 is the second power of the Jacobson radical.

```
BuildHomomorphismFromGradedCap(A, B, phi)
```

Returns the graded homomorphism from the associated graded algebra X of A to the associated graded algebra Y of B, whose cap is ϕ. That is phi is the matrix of the induced homomorphism of $X/Rad^2(X)$ to $Y/Rad^2(Y)$.

```
ChangeIdempotents(A, S)
```

```
ChangeIdempotents(A, S)
```

Returns the basic algebra isomorphic to A, obtained by permuting the order of the idempotents by the permutation S. The permutation S can be given as an element of the symmetric group or as the sequence of images of the permutation.

Example H85E6

In this examples we investigate properties of the basic algebra of a Schur algebra.

```
> A := BasicAlgebraOfSchurAlgebra(3,6,GF(3));
> A;
Basic algebra of dimension 48 over GF(3)
Number of projective modules: 7
Number of generators: 21
> B := BasicAlgebraOfExtAlgebra(A,10);
> B;
Basic algebra of dimension 98 over GF(3)
Number of projective modules: 7
Number of generators: 21
> C := BasicAlgebraOfExtAlgebra(B,10);
> C;
Basic algebra of dimension 48 over GF(3)
Number of projective modules: 7
Number of generators: 21
> boo,mat := IsIsomorphic(A,C);
> boo;
true
> IsAlgebraHomomorphism(mat);
true
```

So we see that A is isomorphic to its double ext-algebra, and it is graded since the double ext-algebra is graded. Thus we know that the basic algebra A is a Koszul algebra.

Example H85E7

Here we see how to rearrange an algebra by changing the ordering on the primitive idempotents.

```
> A := BasicAlgebraOfSchurAlgebra(3,5,GF(3));
> A;
Basic algebra of dimension 11 over GF(3)
Number of projective modules: 5
Number of generators: 9
> B, uu  := ChangeIdempotents(A,[2,4,5,1,3]);
> B;
Basic algebra of dimension 11 over GF(3)
Number of projective modules: 5
Number of generators: 9
> DimensionsOfProjectiveModules(A);
[ 2, 3, 2, 1, 3 ]
> DimensionsOfProjectiveModules(B);
[ 3, 1, 3, 2, 2 ]
> IsAlgebraHomomorphism(A,B,uu);
true
> uu;
[0 0 0 0 0 0 0 1 0 0 0]
```

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

We see that uu is a permutation matrix.

85.6 Automorphisms and Isomorphisms

Magma has the capability of computing automorphisms and isomorphisms of basic algebras. Because the automorphism groups tend to be rather large, the functions work best on small examples.

`GradedAutomorphismGroupMatchingIdempotents(A)`

Returns the group of graded automorphisms of the basic algebra A that preserve the idempotents of A. Returns also the graded caps (matrices of homomorphisms from $X/Rad^2(X)$ to itself, where X is the Associated Graded algebra of A) of the generators of the automorphism group in two sequenced, nonunipotent generators and unipotent generators.

`GradedAutomorphismGroup(A)`

Returns the group of graded automorphisms of the associated graded algebra X of the basic algebra A. The function returns also the graded caps of the generators of the graded automorphism group. These are the induced automorphisms of $X/Rad^2(X)$ to itself, and they are returned as two lists of nonunipotent and unipotent generators that preserve the idempotents of X and a list of generators that permute the idempotents.

`IsGradedIsomorphic(A, B)`

Returns `true` if the associated graded algebras of A and B are isomorphic, in which case the isomorphism is returned.

`AutomorphismGroupMatchingIdempotents(A)`

Returns the group of all automorphism of the basic algebra A that preserve the basic idempotent structure. That is, any element of this group induces the identity automorphism on the quotient $A/Rad(A)$ of A by its Jacobson radical.

`AutomorphismGroup(A)`

Returns the automorphism group of the basic Algebra A, together with the sequences of nonnilpotent generators preserving idempotents, nilpotent generators preserving idempotents and generators that permute the idempotents.

`IsIsomorphic(A, B)`

Returns `true` if the basic algebra A is isomorphic to the basic algebra B and, if so, the function also returns an isomorphism.

Example H85E8

We compute the automorphism group of a group algebra.

```
> A := BasicAlgebra(SmallGroup(81, 7));
> time ba := AutomorphismGroup(A);
Time: 6.260
> #ba;
343585013821340887357640753177080848468071681334
> Factorization(#ba);
[ <2, 1>, <3, 99> ]
```

As expected the automorphism group has a very large unipotent 3-subgroup.

Example H85E9

We can see if small changes in the presentation of an algebra affect the isomorphism type. We define the algebras by generators and relations.

```
> F<e1,e2,z,y,x> := FreeAlgebra(GF(5),5);
> RR:= [(y*z)^3,x^4,x*y*z];
> A := BasicAlgebra(F,RR,2,[<1,2>,<2,1>,<2,2>]);
> A;
Basic algebra of dimension 49 over GF(5)
Number of projective modules: 2
Number of generators: 5
> RS:= [(y*z)^3-x^4,x^5,x*y*z,(z*y)^3];
> B := BasicAlgebra(F,RS,2,[<1,2>,<2,1>,<2,2>]);
> B;
Basic algebra of dimension 49 over GF(5)
Number of projective modules: 2
Number of generators: 5
> RT:= [(y*z)^3-2*x^4,x^5,x*y*z,(z*y)^3];
> C := BasicAlgebra(F,RS,2,[<1,2>,<2,1>,<2,2>]);
> C;
Basic algebra of dimension 49 over GF(5)
Number of projective modules: 2
Number of generators: 5
> time ab, x := IsIsomorphic(A,B);
Time: 0.100
```

```
> ab;
false
> time ac, x := IsIsomorphic(A,C);
Time: 0.050
> print ac;
false
> time bc,x := IsIsomorphic(B,C);
Time: 2.350
> print bc;
true
```

Example H85E10

We can see which groups of order 32 have mod 2 group algebras that are graded. The algebra is graded if and only if it is isomorphic to its associated graded algebra. So we make a list of those algebras whose group algebras are graded. Note that there are 51 isomorphism classes of groups of order 32. The last one (number 51) is elementary abelian and we know that its group algebra is graded.

```
> graded := [];
> for i := 1 to 50 do
>     A := BasicAlgebra(SmallGroup(32,i));
>     B := AssociatedGradedAlgebra(A);
>     boo, map := IsIsomorphic(A, B);
>     if boo then Append(~graded, i); end if;
> end for;
> graded;
[ 1, 2, 3, 5, 9, 12, 14, 16, 18, 21, 22, 25, 36, 39, 45, 46 ]
```

85.7 Modules over Basic Algebras

A module M over a basic algebra B is presented as a sequence of matrices, one for each generator of the algebra.

85.7.1 Indecomposable Projective Modules

The indecomposable projective modules are defined from the structure of the algebra and have associated path trees that solve the homomorphism lifting problem.

```
ProjectiveModule(B, i)
```

The i^{th} projective module of the basic algebra B.

```
PathTree(B, i)
```

The path tree of the i^{th} projective module of the basic algebra B.

`ActionGenerator(B, i)`

The sequence of matrices for the generators of the basic algebra B acting on the i^{th} projective module of B.

`IdempotentActionGenerators(B, i)`

The sequence of matrices for the idempotent generators of the basic algebra B acting on the i^{th} projective module of B.

`NonIdempotentActionGenerators(B, i)`

The sequence of matrices for the nonidempotent generators of the basic algebra B acting on the i^{th} projective module of B.

`Injection(B, i, v)`

Given a vector v in in the i^{th} projective module of the basic algebra B, the function returns the image of inclusion of v into B.

85.7.2 Creation

`AModule(B, Q)`

Given a basic algebra B and a sequence Q of elements in a matrix algebra the function returns the B-module M on which the generators of B act by multiplication by the corresponding elements of Q.

`ProjectiveModule(B, S)`

Given a sequence $S = [s_1, s_2, ...]$, the function returns a projective module which is the direct sum of s_1 copies of the first projective of the algebra B, s_2 copies of the second, etc. It also returns the sequence of inclusions and projections from and to the indecomposable projective modules.

`IrreducibleModule(B, i)`

`SimpleModule(B, i)`

The i^{th} irreducible module of the algebra B. The module is the quotient of the i^{th} projective module by its radical.

`ZeroModule(B)`

The zero B-module.

`RightRegularModule(B)`

The algebra B as a right module over itself. The module is the direct sum of the projectives modules of B.

```
RegularRepresentation(v)
```

If v is an element of a basic algebra given as a vector in the underlying space, then the function computes the matrix of the action by right multiplication of the element on the algebra.

```
Restriction(M, B, xi)
```

If B is a subalgebra of the basic algebra A, ξ is the embedding of B into A, and M is an A-module, then the function returns the restriction of M to a B-module.

```
ChangeAlgebra(M, B, xi)
```

```
ChangeAlgebra(M, B, xi)
```

Given a module M over an algebra A and an algebra homomorphism ξ from B to A, the function returns the module M as a B-module.

```
JacobsonRadical(M)
```

The Jacobson radical of the module M.

```
Socle(M)
```

The socle of the module M. The sum of the simple submodules of M.

85.7.3 Access Functions

```
Algebra(M)
```

Given a module M over a basic algebra B, the function returns B.

```
Dimension(M)
```

The dimension of the module M over its base ring.

```
Action(M)
```

The matrix algebra of the action of the algebra of M on M.

```
IsomorphismTypesOfRadicalLayers(M)
```

Given a module M over a basic algebra, returns the sequence of isomorphism types of simple composition factors in each layer of the radical filtration of M.

```
IsomorphismTypesOfSocleLayers(M)
```

Given a module M over a basic algebra, returns a sequence of isomorphism types of simple composition factors in each socle layer with reversed order, *i. e.* isomorphism types of the socle of M will appear last.

```
IsomorphismTypesOfBasicAlgebraSequence(S)
```

Given a sequence of irreducible modules S for a basic algebra A, return a sequence of isomorphism types comparing with the simple modules of A.

Example H85E11

We show the restriction of a module over an algebra A to a subalgebra of A.

```
> G := SmallGroup(32,7);
> A := BasicAlgebra(G);
> C, mu := Center(A);
> X := RightRegularModule(A);
> Z := JacobsonRadical(X);
> L := Restriction(Z,C,mu);
> L;
AModule L of dimension 31 over GF(2)
> A eq Algebra(L);
True
> IndecomposableSummands(L);
[
    AModule of dimension 1 over GF(2),
    AModule of dimension 30 over GF(2)
]
> Dimension(Socle(L));
16
```

Next we show how to pull back modules along a quotient map. We use the same algebra A.

```
> U := ideal<A|[A.13 +A.17]>;
> Q, theta := quo<A|U>;
> X := ProjectiveModule(Q,1);
> Y := ChangeAlgebras(X,A,theta);
> Y;
AModule Y of dimension 16 over GF(2)
```

Example H85E12

Here is another example of pulling back a module along a quotient map. This one involves algebras with more than one idempotent.

```
> load m11;
Loading "/usr/local/dmagma/libs/pergps/m11"
M11 - Mathieu group on 11 letters - degree 11
Order 7 920 = 2^4 * 3^2 * 5 * 11;  Base 1,2,3,4
Group: G
> A:= BasicAlgebraOfPrincipalBlock(G,GF(2));
> A;
Basic algebra of dimension 22 over GF(2)
Number of projective modules: 3
Number of generators: 9
> DimensionsOfProjectiveModules(A);
[ 8, 8, 6 ]
> I := ideal<A|[A.9]>;
> B, mu := quo<A|I>;
> B;
Basic algebra of dimension 6 over GF(2)
Number of projective modules: 2
Number of generators: 5
> P := ProjectiveModule(B,1);
> P;
AModule P of dimension 3 over GF(2)
> Q := ChangeAlgebras(P,A,mu);
> Algebra(Q) eq A;
true
```

Example H85E13

In this example, we investigate the structure of the projective modules of a basic algebra.

```
> G := PSL(3,3);
> N := Normalizer(G,Sylow(G,2));
> A := BasicAlgebraOfHeckeAlgebra(G,N,GF(2));
> DimensionsOfProjectiveModules(A);
[ 1, 2, 3, 9, 9, 1, 1, 1, 1 ]
> IsomorphismTypesOfRadicalLayers(ProjectiveModule(A,4));
[
    [ 4 ],
    [ 2, 3, 4, 5 ],
    [ 4, 4, 5 ],
    [ 5 ]
]
> IsomorphismTypesOfSocleLayers(ProjectiveModule(A,4));
[
    [ 4 ],
    [ 3, 4, 5 ],
    [ 2, 4, 5 ],
    [ 4, 5 ]
]
```

So we see that, unlike a group algebra, a Hecke algebra can have indecomposable projective modules whose socles are not simple.

85.7.4 Predicates

The following functions return a boolean value.

```
IsSemisimple(M)
```

Returns `true` if the module M is a semisimple module and `false` otherwise. If `true`, then the function also returns a list of the ranks of the primitive idempotents of the algebra. This is also a list of the multiplicities of the simple modules of the algebra as composition factors in a composition series for the module.

`IsProjective(M)`

Returns `true` if the module M is projective. The function also returns a sequence of multiplicities of the standard projective modules as direct summands of the projective cover of M.

`IsInjective(M)`

Returns `true` if the module M is injective. The function also returns a sequence of multiplicities of the standard injective modules as direct summands of the injective hull of M.

85.7.5 Elementary Operations

`m * b`

Given an element b in a basic algebra B and an element m in a module M over B, $m * b$ is the product.

Example H85E14

We obtain the dimensions of the radical layers of the group algebra of an extra special group of order 243 over a field of characteristic 3.

```
> G := ExtraSpecialGroup(3,2);
> G;
Permutation group G acting on a set of cardinality 243
> ff := GF(3);
> A := BasicAlgebra(G,ff);
> A;
Basic algebra of dimension 243 over GF(3)
Number of projective modules: 1
Number of generators: 6
> P := ProjectiveModule(A,1);
> P;
AModule P of dimension 243 over GF(3)
> R := JacobsonRadical(P);
> R;
AModule R of dimension 242 over GF(3)
> while Dimension(R) ne 0 do
>     T := JacobsonRadical(R);
>     print Dimension(R) - Dimension(T);
>     R := T;
> end while;
4
11
20
30
36
39
```

```
36
30
20
11
4
1
```

Example H85E15

We consider the mod-2 group algebra of an extraspecial group of order 128, and construct the module induced from the trivial module on the subgroups of order 4 generated by the first generator of the group.

```
> G := ExtraSpecialGroup(2,3);
> G;
Permutation group G acting on a set of cardinality 128
> F := GF(2);
> A := BasicAlgebra(G,F);
> A;
Basic algebra of dimension 128 over GF(2)
Number of projective modules: 1
Number of generators: 8
> A.1;
(1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
    0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
    0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
    0 0 0 0 0 0 0 0 0 0 0 0)
```

Note that A.1 is the unique idempotent (identity element) in the group algebra, whereas A.2 is G.1-1 where G.1 is the first generator of the group.

```
> A.2;
(0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
    0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
    0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
    0 0 0 0 0 0 0 0 0 0 0 0)
> g := A.1+A.2;
```

Now we check the order of g.

```
> g^2;
(1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
    0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0
    0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
    0 0 0 0 0 0 0 0 0 0 0 0)
> g^4;
(1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
    0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
    0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
    0 0 0 0 0 0 0 0 0 0 0 0)
```

```
> g^4 eq A!1;
true
```

So g has order 4.

```
> P := ProjectiveModule(A,1);
> P;
AModule P of dimension 128 over GF(2)
```

Note that P is generated by P.1 which corresponds to the identity element of A if we think of P as the algebra A as a module over itself. Now we create the induced module as the submodule generated by $(g - 1)^3$, since $(g - 1)^4 = 0$.

```
> U := sub<P|P.1*A.6>;
> U;
AModule U of dimension 32 over GF(2)
```

Because the dimension is a quarter of the order of the group we can be sure that we have the right thing by just checking that U is generated by a g fixed point.

```
> U.1*g eq U.1;
true
```

85.8 Homomorphisms of Modules

A homomorphism from module M to module N is simply a matrix that commutes with the action of the algebra on M and N.

85.8.1 Creation

`AHom(M, N)`

The space of homomorphisms from module M to module N.

`PHom(M,N)`

The space of projective homomorphisms from module M to module N. That is, the space of all homomorphisms that factor through a projective module.

`ZeroMap(M, N)`

The zero homomorphism from module M to module N.

`LiftHomomorphism(x, n)`

Given an element x in a module over a basic algebra and a natural number n, the function returns the homomorphism from the n^{th} projective module for the algebra to the module with the property that the idempotent e of the projective module maps to $x * e$.

```
LiftHomomorphism(X, N)
```

Given a sequence $X = [x_1, \ldots, x_t]$ of elements in a module M over a basic algebra and a sequence $N = [n_1, \ldots, n_s]$ of nonnegative integers, such that $n_1 + \ldots + n_s = t$, the function returns the homomorphism $P \to M$ from the projective module $P = \sum_{j=1}^{s} P_j^{n_i}$ to M that takes the idempotent e for the i^{th} summand in P to the element $x_i * e$ in M. Here P_j denotes the j^{th} projective module for the algebra.

```
Pushout(M, f1, N1, f2, N2)
```

The pushout of the diagram

$$\begin{tikzcd} M \arrow[r, "f_1"] \arrow[d, "f_2"] & N_1 \\ N_2 & \end{tikzcd}$$

The function returns the module $L = (N_1 \oplus N_2)/\{(f_1(m), -f_2(m)) | m \in M\}$ and the homomorphisms $g_1 : N_1 \longrightarrow L$ and $g_2 : N_2 \longrightarrow L$ such that $f_1 g_1 = f_2 g_2$.

```
Pullback(f1, M1, f2, M2, N)
```

The pullback of the diagram

$$\begin{tikzcd} & M_2 \arrow[d, "f_2"] \\ M_1 \arrow[r, "f_1"] & N \end{tikzcd}$$

The function returns the module $L = \{(m_1, m_2) \in M_1 \oplus M_2 | f_1(m_1) = f_2(m_2)\}$ and the homomorphisms $g_1 : L \longrightarrow M_1$ and $g_2 : L \longrightarrow M_2$ such that $g_1 f_1 = g_2 f_2$.

85.8.2 Access Functions

```
IsModuleHomomorphism(f)
```

Returns `true` if the map f is a homomorphism of modules over the algebra.

```
Domain(f)
```

The domain of f.

```
Codomain(f)
```

The codomain of f.

```
Kernel(f)
```

The kernel of f and the inclusion of the kernel in `Domain(f)`.

```
Cokernel(f)
```

The cokernel of f and the quotient map from `Codomain(f)` onto the cokernel.

85.8.3 Projective Covers and Resolutions

A projective cover of a module M is a projective module P and a surjective homomorphism $\phi : P \longrightarrow M$ such that P is minimal with respect to the property of having such a surjective homomorphism to M. A projective resolution to n steps of an A-module M is a pair consisting of a complex

$$P_n \xrightarrow{\partial_n} P_{n-1} \longrightarrow \ldots \longrightarrow P_1 \xrightarrow{\partial_1} P_0$$

which is exact except at the ends, and an augmentation homomorphism $\epsilon : P_0 \to M$ that is a projective cover of M. In addition, the image of ∂_1 must equal the kernel of ϵ. The resolution is minimal if each P_i is a projective cover of its image in P_{i-1}. In this case the i^{th} syzygy module is the image of ∂_i.

In the implementation the main function is `CompactProjectiveResolution.` This function computes a minimal projective resolution of a given module and stores the minimal amount of information that is necessary to create the boundary maps and the terms of the resolution. It runs relatively fast because it avoids the computation of the terms of the projective resolution as modules over the algebra. Instead the terms in the compact resolution are only vector spaces together with a sequence of types for the projective modules. The other information that is recorded is the sequence of images of the generators for the indecomposable projective modules. That is, for the boundary map

$$P_n \xrightarrow{\partial_n} P_{n-1}$$

the module $P_n \cong \oplus_{i=1}^m Q_i$ where each Q_i is an indecomposable projective module generated by an element a_i corresponding to the appropriate idempotent in the basic algebra. The function records the images $\partial_n(a_i)$ as a sequence of vectors in the vector space of the module P_{n-1}.

`ProjectiveCover(M)`

The projective cover of the module M given as the projective module P, the surjective homomorphism of P onto M, the sequences of inclusion and projection homomorphism of P from and to its indecomposable direct summands and the isomorphism type of P in the form of a list of the number of copies of the projective modules of the algebra of each type that make up P.

`ProjectiveResolution(M, n)`

The complex giving the minimal projective resolution of the module M out to n steps together with the augmentation homomorphism from the projective cover of M into M. Note that homomorphisms go from left to right so that last term of the complex (in degree 0) is the projective cover of M and the cokernel of the last homomorphism in the complex is M. The complex is constructed from the compact projective resolution of M. The function creates the compact projective resolution if it has not already been computed.

```
CompactProjectiveResolution(M, n)
```

Returns a minimal projective resolution for the module M out to n steps in compact form together with the augmentation map ($P_0 \to M$). The compact form of the resolution is a list of the minimal pieces of information needed to reconstruct the boundary maps in the resolution. That is the boundary map ($P_i \xrightarrow{\partial_i} P_{i-1}$) is recorded as a matrix whose entries are the images of the generators for indecomposable projective modules making up P_i in the indecomposable projective modules making up P_{i-1}. If a compact projective resolution has been previously computed to degree m and $m < n$ then the function extends that resolution by $n - m$ steps. If $m \geq n$ the function returns the previously computed compact projective resolution. The function returns a record with the fields:

(a) The list of isomorphism types of the projective modules in the resolution, each given as a sequence of integers giving the number of direct summands of each indecomposable projective in the module (field name `BettiNumbers`).

(b) The record of the boundary maps (field name `ResolutionRecord`).

(c) The module M (field name `Module`).

(d) The augmentation map (field name `AugmentationMap`).

(e) The type of the resolution, whether projective or injective (field name `Typ`).

```
CompactProjectiveResolutionsOfSimpleModules(A,n)
```

Returns a sequence of the compact projective resolutions of the simple A-modules computed to degree n.

```
SyzygyModule(M, n)
```

The n^{th} syzygy module of the module M. The module is constructed from the compact projective resolution of M. The compact resolution is constructed if it does not already exist.

```
SimpleHomologyDimensions(M)
```

The sequence of sequences of dimensions of the homology groups $\mathrm{Tor}_j(S_i, M)$ for simple modules S_i and module M, to the extent that they have been computed.

Example H85E16

We consider the basic algebra of a quiver with relation. The quiver has four nodes and 5 non-idempotent generators (a, b, c, d, e). The first goes from node 1 to node 2, the second from 2 to 3, the third from 3 to 4, the fourth from 3 to 2 and the last from 4 to 1. They satisfy the relations $bcfadbd = (abcf)^5ab = (bd)^2b = 0$.

```
> ff := GF(8);
> FA<e1,e2,e3,e4,a,b,c,d,f> := FreeAlgebra(ff,9);
> rrr := [b*c*f*a*b*d*b,a*b*c*f*a*b*c*f*a*b*c*f*a*b*c*f*a*b*c*f*a*b,
>     b*d*b*d*b];
> BA := BasicAlgebra(FA,rrr,4,[<1,2>,<2,3>,<3,4>,<3,2>,<4,1>]);
> BA;
Basic algebra of dimension 296 over GF(2^3)
Number of projective modules: 4
Number of generators: 9
```

Now we take the projective resolutions of the simple modules out to 5 steps. We print the type of the projective module at each stage.

```
> for i := 1 to 4 do
>     S := SimpleModule(BA,i);
>     prj := CompactProjectiveResolution(S, 5);
>     SimpleHomologyDimensions(S);
> end for;
[
    [ 0, 0, 10, 0 ],
    [ 0, 0, 5, 0 ],
    [ 0, 0, 2, 0 ],
    [ 0, 0, 1, 0 ],
    [ 0, 1, 0, 0 ],
    [ 1, 0, 0, 0 ]
]
[
    [ 0, 0, 16, 0 ],
    [ 0, 0, 8, 0 ],
    [ 0, 0, 4, 0 ],
    [ 0, 0, 2, 0 ],
    [ 0, 0, 1, 0 ],
    [ 0, 1, 0, 0 ]
]
[
    [ 0, 0, 0, 0 ],
    [ 0, 0, 0, 0 ],
    [ 0, 0, 0, 0 ],
    [ 0, 0, 0, 0 ],
    [ 0, 1, 0, 1 ],
    [ 0, 0, 1, 0 ]
]
[
    [ 0, 0, 0, 0 ],
    [ 0, 0, 0, 0 ],
    [ 0, 0, 0, 0 ],
    [ 0, 0, 0, 0 ],
    [ 1, 0, 0, 0 ],
    [ 0, 0, 0, 1 ]
]
```

So we see that the third and fourth simple modules have finite projective dimension. The projective resolutions of the first and second simple modules appear to have exponential rates of growth but the terms after the second term are all direct sums of copies of the third projective module.

```
> for i := 1 to 4 do
>     Dimension(Socle(ProjectiveModule(BA,i)));
> end for;
12
13
25
12
```

Notice that the socles of the projective modules have very large dimensions so the injective resolutions are probably going to grow at a very rapid rate.

Example H85E17

We create the quotient of the group algebra of a p-group by the ideal generated by a central element. In particular, we choose the group algebra of an extra special 3-group and factor out the ideal generated by $(z-1)^2$ where z is a central element of order 3. Then we compare the size of the projective resolution of the trivial module for the new algebra with that of the antecedent group algebra.

```
> G := ExtraSpecialGroup(3,1);
> F := GF(3);
> B := BasicAlgebra(G,F);
> B;
Basic algebra of dimension 27 over GF(3)
Number of projective modules: 1
Number of generators: 4
> s := NonIdempotentGenerators(B)[3];
```

Now check that `s` is in the center.

```
> [s*x eq x*s: x in Generators(B)];
[ true, true, true, true ]
> P := ProjectiveModule(B,1);
> Q := quo<P|P.1*s^2>;
> Q;
AModule Q of dimension 18 over GF(3)
```

We need to create the path tree for the projective module of the matrix algebra of the action on `Q`. In this case it is an easy exercise because the last 9 element of the basis of the projective module for `B` span the submodule that we are factoring out. This can actually be seen from the path tree for the projective module of `B`.

```
> PathTree(B,1);
[ <1, 1>, <1, 2>, <2, 2>, <1, 3>, <2, 3>, <3, 3>, <4, 3>,
<5, 3>, <6, 3>, <1, 4>, <2, 4>, <3, 4>, <4, 4>, <5, 4>,
<6, 4>, <7, 4>, <8, 4>, <9, 4>, <10, 4>, <11, 4>, <12, 4>,
<13, 4>, <14, 4>, <15, 4>, <16, 4>, <17, 4>, <18, 4> ]
```

So we get the path tree for the new module by truncation.

```
> PT := [PathTree(B,1)[j]: j in [1 .. 18]];
> PT;
[ <1, 1>, <1, 2>, <2, 2>, <1, 3>, <2, 3>, <3, 3>,
<4, 3>, <5, 3>, <6, 3>, <1, 4>, <2, 4>, <3, 4>,
<4, 4>, <5, 4>, <6, 4>, <7, 4>, <8, 4>, <9, 4> ]
```

Now form the new basic algebra.

```
> C := BasicAlgebra([<Action(Q),PT>]);
> C;
Basic algebra of dimension 18 over GF(3)
Number of projective modules: 1
Number of generators: 4
> S := SimpleModule(C,1);
> prj := CompactProjectiveResolution(S, 15);
> SimpleHomologyDimensions(S);
[ 92, 77, 70, 57, 51, 40, 35, 26, 22, 15, 12, 7, 5, 2, 1 ]
```

Now compare this with the projective resolution for the group algebra.

```
> T := SimpleModule(B,1);
> pj2 := CompactProjectiveResolution(T,15);
> SimpleHomologyDimensions(T);
[ 20, 18, 17, 16, 15, 14, 12, 10, 9, 8, 7, 6, 4, 2, 1 ]
```

85.9 Duals and Injectives

If k is the base ring for the algebra A then the k-dual $\mathrm{Hom}_k(M, k)$ for a right module M over A is a right module over the opposite algebra OA of A. Furthermore, the dual of a projective OA-module is an injective A-module and the dual of a projective OA-resolution of a module M is an A-injective resolution of the dual of M.

```
Dual(M)
```

Given a module M defined over a basic algebra M, this function returns the dual of M as a module over the opposite of the algebra of M. Note that the opposite of the algebra of M is created if it does not already exist.

```
BaseChangeMatrix(A)
```

Given a basic algebra A that has opposite algebra O, the function creates the change of basis matrix B from the vector space of A to the vector space of O, so that if x, y are in A then $(xy)B$ is the same as $(yB)(xB) \in O$.

85.9.1 Injective Modules

Injective hulls, and injective resolutions of a module are computed by taking the projective cover or projective resolution of the dual module over the opposite algebra and then again taking the dual to retrieve modules or complexes over the original algebra. If the opposite algebra of the module has not been computed then it will be created in the evaluation of any of the injective module functions.

```
InjectiveModule(B, i)
```

The i^{th} injective module of the algebra B.

```
InjectiveHull(M)
```

The injective hull of the module M is the injective module I of minimal dimension such that there is an inclusion $\iota : M \to I$. The function returns I, ι, the sequences of inclusions and projections from and to the indecomposable injective summands of I, and the type of I as a sequence $T := [t_1 \dots, t_s]$ where I has t_1 summands of type 1, t_2 of type 2, etc.

```
InjectiveResolution(M, n)
```

The complex giving the minimal injective resolution of the module M together with the inclusion homomorphism from M into its injective hull. Note that homomorphisms go from left to right so that the kernel of the first homomorphism in the complex is M. The function computes the compact injective resolution and creates the complex of the injective resolution from that.

```
CompactInjectiveResolution(M, n)
```

A minimal injective resolution for the module M out to n steps in compact form together with the coaugmentation map ($M \to I_0$). The compact form of the resolution is a list of the minimal pieces of information needed to reconstruct the boundary maps in the resolution. That is, the boundary map ($I_{i-1} \xrightarrow{\partial_i} I_i$) is recorded as a matrix whose entries are the images of the generators for indecomposable injective modules making up I_{i-1} in the indecomposable projective modules making up I_i. The actual return of the function is the compact projective resolution of the dual module of M over the opposite algebra of the algebra of M. The return is a record with the fields:

(a) The list of isomorphism types of the injective modules in the resolution, each given as a sequence of integers giving the number of direct summands of each indecomposable injective in the module (field name `BettiNumbers`).

(b) The record of the boundary maps (field name `ResolutionRecord`).

(c) The module M (field name `Module`).

(d) The coaugmentation map (field name `CoaugmentationMap`).

(e) The type of the resolution, whether projective or injective (field name `Typ`).

`InjectiveSyzygyModule(M, n)`

The n^{th} injective-syzygy module of M. The module is constructed from the compact injective resolution of M. The compact resolution is constructed if it does not already exist.

`SimpleCohomologyDimensions(M)`

The sequence of sequences of dimensions of the cohomology groups $\mathrm{Ext}^j(S_i, M)$ for simple modules S_i and module M, to the extent that they have been computed.

Example H85E18

We create the basic algebra of a quiver over a field with 8 elements. The quiver has two nodes and three arrows, going from node 1 to node 2, from 2 to 1 and from 1 to 1. The relations are given in the sequence `rrr`.

```
> ff := GF(8);
> FA<e1,e2,a,b,c> := FreeAlgebra(ff,5);
> rrr := [a*b*a*b*a, c*c*c*c, a*b*c - c*a*b];
> B := BasicAlgebra(FA,rrr,2,[<1,2>,<2,1>,<1,1>]);
> B;
Basic algebra of dimension 41 over GF(2^3)
Number of projective modules: 2
Number of generators: 5
> DimensionsOfProjectiveModules(B);
[ 20, 21 ]
> DimensionsOfInjectiveModules(B);
[ 24, 17 ]
> P1 := ProjectiveModule(B,1);
> Socle(P1);
AModule of dimension 1 over GF(2^3)
```

We consider the injective resolution of the first projective module.

```
> time in1 := CompactInjectiveResolution(P1,10);
reverse trees
Time: 3.850
```

Note that part of the time was required to create the opposite algebra of `B`.

```
> SimpleCohomologyDimensions(P1);
[
    [ 1, 0 ],
    [ 0, 4 ],
    [ 4, 0 ],
    [ 4, 0 ],
    [ 4, 0 ],
    [ 4, 0 ],
    [ 4, 0 ],
    [ 4, 0 ],
    [ 4, 0 ],
    [ 4, 0 ]
]
```

The injective resolution appears to be periodic. Now we look at a module constructed from the resolution.

```
> M := InjectiveSyzygyModule(P1,6);
> M;
AModule M of dimension 64 over GF(2^3)
```

Consider the space of endomorphisms of M.

```
> hh := AHom(M,M);
> hh;
KMatrixSpace of 64 by 64 matrices and dimension 128 over GF(2^3)
> [Rank(hh.i): i in [1 .. Dimension(hh)]];
[ 16, 16, 16, 16, 12, 12, 12, 12, 8, 8, 8, 8, 8, 8, 8, 8, 6, 6, 6, 6, 4, 4, 4,
4, 4, 4, 4, 4, 2, 2, 2, 2, 4, 8, 12, 16, 16, 12, 8, 4, 8, 4, 12, 16, 4, 8, 12,
16, 16, 16, 16, 16, 2, 4, 6, 8, 8, 6, 4, 2, 4, 2, 6, 8, 2, 4, 6, 8, 12, 12, 12,
12, 8, 8, 8, 8, 8, 8, 8, 8, 6, 6, 6, 6, 4, 4, 4, 4, 4, 4, 4, 4, 2, 2, 2, 2, 16,
16, 16, 16, 12, 12, 12, 12, 8, 8, 8, 8, 8, 8, 8, 8, 6, 6, 6, 6, 4, 4, 4, 4, 4,
4, 4, 4, 2, 2, 2, 2 ]
```

Note that no generator of the endomorphism ring has rank more than 16. This would indicate that the module is decomposable since the identity map must be a sum of homomorphisms of smaller rank.

How can we produce a decomposition? One method is the following.

```
> vv := Random(hh);
> Rank(vv);
64
> vv*vv eq vv;
false
> [Rank(vv*vv-u*vv):u in ff];
[ 60, 64, 64, 64, 64, 64, 64, 64 ]
> [u:u in ff];
[ 1, ff.1, ff.1^2, ff.1^3, ff.1^4, ff.1^5, ff.1^6, 0 ]
> Rank(vv*vv - vv);
60
> U := vv*vv - vv;
> Rank(U);
60
> Rank(U*U);
56
> Rank(U*U*U);
52
> Rank(U*U*U*U);
48
> Rank(U*U*U*U*U);
48
```

```
> Rank(U*U*U*U*U*U);
48
> T := U*U*U*U;
> N1 := Kernel(T);
> N2 := Image(T);
> Dimension(N1);
16
> Dimension(N2);
48
> Dimension(N1+N2);
64
```

So the sum of N1 and N2 must be all of M and by counting dimensions, it must be a direct sum.

85.10 Cohomology

`CohomologyRingGenerators(P)`

Given a compact projective resolution P for a simple module S over a basic algebra A, the function returns the chain maps in compact form of a minimal set of generators for the cohomology $\mathrm{Ext}^*_A(S,S)$, as well as some other information. The record that is returned has the following fields:

(a) The list of maps in compact form for the chain map of the generators (field name ChainMapRecord).

(b) The sequence of degrees of cohomology generators (field name ChainDegrees).

(c) The tops of the chain maps (maps on modules modulo radicals) for the purposes of computing products (field name TopsOfCohomologyGenerators).

(d) The tops of the chain maps representing monomials in the generators (field name TopsOfCohomologyChainMaps).

(e) The original compact projective resolution (field name ProjectiveResolution).

`CohomologyRightModuleGenerators(P, Q, CQ)`

Given projective resolutions P and Q for simple modules S and T over a basic algebra A and the cohomology generators CQ for T associated to the resolution Q, the function returns the chain maps in compact form of the minimal generators for the cohomology $\mathrm{Ext}^*_A(S,T)$ as a right module over the cohomology ring $\mathrm{Ext}^*_A(T,T)$. The function returns a record consisting of the following fields.

(a) The list of maps in compact form for the chain map of each cohomology generator (field name ChainMapRecord).

(b) The sequence of degrees of cohomology generators (field name ChainDegrees).

(c) The tops of the chain maps (maps on modules module radicals) for the purposes of computing products (field name TopsOfCohomologyGenerators).

`CohomologyLeftModuleGenerators(P, CP, Q)`

Given projective resolutions P and Q for simple modules S and T over a basic algebra A and the cohomology generators CP for T associated to the resolution Q, the function returns the chain maps in compact form of the minimal generators for the cohomology $\mathrm{Ext}^*_A(S,T)$ as a left module over the cohomology ring $\mathrm{Ext}^*_A(S,S)$. The function returns a record consisting of the following fields.

(a) The list of maps in compact form for the chain map of each cohomology generator (field name `ChainMapRecord`).

(b) The sequence of degrees of cohomology generators (field name `ChainDegrees`).

(c) The tops of the chain maps (maps on modules module radicals) for the purposes of computing products (field name `TopsOfCohomologyGenerators`).

`DegreesOfCohomologyGenerators(C)`

Given the generators C for cohomology, as either module generators or as ring generators, the function returns the list of degrees of the minimal generators.

`CohomologyGeneratorToChainMap(P, Q, C, n)`

Given the projective resolutions P and Q of two modules M and N and the cohomology generators C of the cohomology module, $\mathrm{Ext}^*_B(M,N)$, the function returns the chain map from P to Q that lifts the n^{th} generator of the cohomology module and has degree equal to the degree of that generator.

`CohomologyGeneratorToChainMap(P, C, n)`

Given the projective resolution P of a module and the cohomology generators C of the cohomology ring of that module, the function returns the chain map from P to P that lifts the n^{th} generator of the cohomology ring and has degree equal to the degree of that generator.

Example H85E19

We create the Basic algebra for the principal block of the sporadic simple group M_{11} in characteristic 2. The block algebra has three simple modules of dimension 1, 44, and 10. The basic algebra has dimension 22.

```
> ff := GF(2);
> VV8 := VectorSpace(ff,8);
> BB8 := Basis(VV8);
> MM8 := MatrixAlgebra(ff,8);
> e11 := MM8!0;
> e12 := MM8!0;
> e13 := MM8!0;
> e11[1]  := BB8[1];
> e11[4]  := BB8[4];
> e11[5]  := BB8[5];
> e11[8]  := BB8[8];
> e12[2]  := BB8[2];
> e12[7]  := BB8[7];
> e13[3]  := BB8[3];
> e13[6]  := BB8[6];
> a1 := MM8!0;
> b1 := MM8!0;
> c1 := MM8!0;
> d1 := MM8!0;
> e1 := MM8!0;
> f1 := MM8!0;
> a1[1] :=  BB8[2];
> a1[5] :=  BB8[7];
> b1[1] :=  BB8[3];
> b1[4] :=  BB8[6];
> c1[2] :=  BB8[4];
> c1[7] :=  BB8[8];
> e1[3] :=  BB8[5];
> e1[6] :=  BB8[8];
> f1[3] :=  BB8[6];
> A1 := sub< MM8 | [e11, e12, e13, a1, b1, c1, d1, e1, f1] >;
> T1 := [ <1,1>,<1,4>,<1,5>,<2,6>,<3,8>,<4,5>,<5,4>,<6,8>];
> VV6 := VectorSpace(ff,6);
> BB6 := Basis(VV6);
> MM6 := MatrixAlgebra(ff,6);
> e21 := MM6!0;
> e22 := MM6!0;
> e23 := MM6!0;
> e22[1]  := BB6[1];
> e22[5]  := BB6[5];
> e22[6]  := BB6[6];
> e21[2]  := BB6[2];
> e21[4]  := BB6[4];
> e23[3]  := BB6[3];
> a2 := MM6!0;
> b2 := MM6!0;
> c2 := MM6!0;
> d2 := MM6!0;
> e2 := MM6!0;
> f2 := MM6!0;
> a2[4] := BB6[6];
> b2[2] := BB6[3];
> c2[1] := BB6[2];
> d2[1] := BB6[5];
> d2[5] := BB6[6];
> e2[3] := BB6[4];
> A2 := sub< MM6 | [e21, e22, e23, a2, b2, c2, d2, e2, f2]>;
> T2 := [ <1,2>, <1,6>, <2,5>, <3,8>, <1,7>, <5,7> ];
> VV8 := VectorSpace(ff,8);
> BB8 := Basis(VV8);
> MM8 := MatrixAlgebra(ff,8);
> e31 := MM8!0;
> e32 := MM8!0;
> e33 := MM8!0;
> e31[2]  := BB8[2];
> e31[6]  := BB8[6];
> e32[4]  := BB8[4];
> e33[1]  := BB8[1];
> e33[3]  := BB8[3];
> e33[5]  := BB8[5];
> e33[7]  := BB8[7];
> e33[8]  := BB8[8];
> a3 := MM8!0;
> b3 := MM8!0;
> c3 := MM8!0;
> d3 := MM8!0;
> e3 := MM8!0;
> f3 := MM8!0;a3[2] :=  BB8[4];
> b3[6] :=  BB8[8];
> b3[2] :=  BB8[7];
> c3[4] :=  BB8[6];
> e3[1] :=  BB8[2];
> e3[3] :=  BB8[6];
> f3[1] :=  BB8[3];
> f3[3] :=  BB8[5];
> f3[5] :=  BB8[7];
> f3[7] :=  BB8[8];
> A3 := sub< MM8 | [e31, e32, e33, a3, b3, c3, d3, e3, f3] >;
> T3 := [ <1,3>,<1,8>,<1,9>,<2,4>,<3,9>,<4,6>,<5,9>,<6,5>];
>
> m11 := BasicAlgebra( [<A1, T1>, <A2, T2>, <A3, T3>] );
> m11;
Basic algebra of dimension 22 over GF(2)
Number of projective modules: 3
Number of generators: 9
> s1 := SimpleModule(m11,1);
> s2 := SimpleModule(m11,2);
```

Now we compute the projective resolutions of the first and second simple modules. Then we find the degrees of their cohomology ring generators.

```
> prj1 := CompactProjectiveResolution(s1,20);
> prj2 := CompactProjectiveResolution(s2,20);
> CR1 := CohomologyRingGenerators(prj1);
> CR2 := CohomologyRingGenerators(prj2);
> DegreesOfCohomologyGenerators(CR1);
[ 3, 4, 5 ]
> DegreesOfCohomologyGenerators(CR2);
```

```
[ 1, 2 ]
```

Finally we look at the cohomology Ext(s2, s1) as a left module over the cohomology ring of s1 and as a right module over the cohomology ring of s2.

```
> CR12 := CohomologyLeftModuleGenerators(prj1,CR1,prj2);
> DegreesOfCohomologyGenerators(CR12);
[ 1, 2, 3, 4 ]
> CR12 := CohomologyRightModuleGenerators(prj1,prj2,CR2);
> DegreesOfCohomologyGenerators(CR12);
[ 1 ]
```

So as a module over the cohomology ring of s1 it is generated by 4 elements. But as a module over the cohomology ring of s2 it is generated by a single element.
Next we get the chain complex for the projective resolution of the first simple module and the chain map for the third generator of the cohomology ring of the first simple module.

```
> pj1 := ProjectiveResolution(s1,20);
> pj1;
Basic algebra complex with terms of degree 20 down to 0
Dimensions of terms: 74 66 68 68 60 54 54 54 48 40 40 42 34 26 28 28 20 14 14
   14 8
> gen113 := CohomologyGeneratorToChainMap(pj1,CR1,3);
> gen113;
Basic algebra chain map of degree -5
```

We can compose this with itself.

```
> gen113*gen113;
Basic algebra chain map of degree -10
```

Now compute the kernel and the dimensions of the homology of the kernel.

```
> Ker, phi := Kernel(gen113);
> Ker, phi;
Basic algebra complex with terms of degree 20 down to 0
Dimensions of terms: 20 15 19 20 20 17 17 20 22 15 17 22 20 15 19 20 20 14 14
14 8
Basic algebra chain map of degree 0
> DimensionsOfHomology(Ker);
[ 0, 0, 0, 2, 0, 0, 0, 2, 0, 0, 0, 2, 0, 0, 0, 3, 0, 0, 0 ]
```

We apply the same procedure in the case of the cokernel.

```
> Cok, mu := Cokernel(gen113);
> Cok, mu;
Basic algebra complex with terms of degree 20 down to 0
Dimensions of terms: 74 66 68 68 60 0 3 5 0 0 3 5 0 0 3 5 0 0 3 5 0
Basic algebra chain map of degree 0
> DimensionsOfHomology(Cok);
```

```
[ 0, 0, 0, 27, 0, 0, 2, 0, 0, 0, 2, 0, 0, 0, 2, 0, 0, 0, 2 ]
```

We can also check the image.

```
> Imm, theta, gamma := Image(gen113);
> Imm;
Basic algebra complex with terms of degree 20 down to 0
Dimensions of terms: 0 0 0 0 0 54 51 49 48 40 37 37 34 26 25 23 20 14 11 9 8
> DimensionsOfHomology(Imm);
[ 0, 0, 0, 0, 27, 0, 0, 2, 0, 0, 0, 2, 0, 0, 0, 2, 0, 0, 0 ]
```

We can check that certain things make sense.

```
> IsChainMap(theta);
true
> IsChainMap(gamma);
true
```

85.10.1 Ext-Algebras

The ext-algebra of an algebra B is the algebra $Ext^*_B(S,S)$ for $S = S_1 \oplus \ldots \oplus S_n$ where $S_1, \ldots, S_n$ are all of the simple B-modules. In the event that the algebra B had finite global dimension, this is a finite dimensional algebra and we can form its basic algebra.

```
ExtAlgebra(A, n)
```

The function computes the information on the ext-algebra B of the basic algebra A where the projecive resolutions and cohomology have been computed to degree n. The function returns a record carrying the data:

(i) A free algebra F,

(ii) A list of relations in the elements of F, such that the ext-algebra is the quotient F/I where I is the ideal generated by the relations,

(iii) The sequence of chain maps of the generators. Each chain map goes from the projective resolution of one simple module to that of another.

(iv) The seqence of degrees of the generators.

(v) A sequence of sequences of sequences of basis element such the j^{th} element of the i^{th} sequence is a basis of in $Ext^*_A(S_i, S_j)$ where S_i is the i^{th} simple module.

(vi) A basis of the entire ext-algebra, the concatenation of the previous sequences.

(vii) The number of steps of the cohomology that were computed.

(viii)The global dimension that has been computed. This number is smaller than the number of steps only in the case that the global dimension of A is less than n, meaning that the n^{th} step in the projective resolution of any simple module is the zero module.

```
BasicAlgebraOfExtAlgebra(ext)
```

The function creates the basic algebra from a computed ext-algebra. The input is the output of the `ExtAlgebra` function. If the ext-algebra is not verified to be finite dimensional by the computation, then an error is returned.

```
BasicAlgebraOfExtAlgebra(A)
```

The function forms the basic algebra from a computed ext-algebra of the basic algebra A. If no ext-algebra for A has been computed or if the ext-algebra is not verified to be finite dimensional then an error is returned.

```
BasicAlgebraOfExtAlgebra(A, n)
```

The function creates the basic algebra for the ext-algebra of A computed to n steps. If no ext-algebra for A to n steps has been computed then it computes one. If the ext-algebra is not verified to be finite dimensional by the computation, then an error is returned.

```
SumOfBettiNumbersOfSimpleModules(A, n)
```

This function computes the Betti numbers of all of simple A-modules out to degree n. This is the dimension of the ext-algebra of A computed to degree n.

Example H85E20

We construct the basic algebra of the algebra B of lower triangular matrices over the field with 5 elements. Note that because the identity element of B is the sum of primitive idempotents of rank 1, B is actually isomorphic to its basic algebra.

```
> A := MatrixAlgebra(GF(5),10);
> U := A!0;
> ElementaryMatrix := function(i,j);
>     W := U;
>     W[i,j] := 1;
>     return W;
> end function;
> S := &cat[[ElementaryMatrix(i,j): i in [j .. 10]]:j in [1 .. 10]];
> B := sub<A|S>;
> B;
Matrix Algebra of degree 10 with 55 generators over GF(5)
> C := BasicAlgebra(B);
> C;
Basic algebra of dimension 55 over GF(5)
Number of projective modules: 10
Number of generators: 19
```

Note that C has the same dimension as B. The two are isomorphic, though they have different types.

```
> SumOfBettiNumbersOfSimpleModules(C,9);
19
```

```
> SumOfBettiNumbersOfSimpleModules(C,10);
19
```

From the above we see that C has global dimension at most 9. Consequently, we can compute the basic algebra of its ext-algebra.

```
> D:= ExtAlgebra(C,10);
> E := BasicAlgebraOfExtAlgebra(D);
> E;
Basic algebra of dimension 19 over GF(5)
Number of projective modules: 10
Number of generators: 19
> SumOfBettiNumbersOfSimpleModules(E,8);
54
> SumOfBettiNumbersOfSimpleModules(E,9);
55
> SumOfBettiNumbersOfSimpleModules(E,10);
55
```

Here we see that E has global dimension 9. So we compute the basic algebra of its ext-algebra.

```
> F := BasicAlgebraOfExtAlgebra(E,10);
> F;
Basic algebra of dimension 55 over GF(5)
Number of projective modules: 10
Number of generators: 19
> G := BasicAlgebraOfExtAlgebra(F,10);
> G;
Basic algebra of dimension 19 over GF(5)
Number of projective modules: 10
Number of generators: 19
```

So it would appear that C and F are isomorphic as well as E and G.

85.11 Group Algebras of p-groups

There is a special type for the basic algebras which are the modular group algebras of p-groups for p a prime. If G is a finite p and k is a field of characteristic p, then the commands `BasicAlgebra(`G, k`)` and `BasicAlgebra(`G`)` automatically create a basic algebra of type `AlgBasGrpP`. The type is optimized for the computation of cohomology rings. Included for this type are restriction and inflation maps. Most of the functions for modules and complexes are the same as for general basic algebras.

85.11.1 Access Functions

`Group(A)`

The group which defines the algebra A.

`PCGroup(A)`

The internal PC group of the algebra A.

`PCMap(A)`

The map from `Group(A)` to `PCGroup(A)` for an algebra A.

`AModule(M)`

Converts a `GModule` M over a p-group to a module over the basic algebra of that group.

`GModule(M)`

Returns the standard module of the algebra A as a module over `Group(A)` and as a module over `PCGroup(A)`.

`GModule(M)`

Converts a module M for the basic algebra of a p-group into a module over the p-group.

85.11.2 Projective Resolutions

`ResolutionData(A)`

Returns the data needed to compute the projective resolution of an A-module for an algebra A. The data is given as a record with the fields:

(a) The matrices of the `PCGenerators` of the p-group on the standard indecomposable projective module for the algebra (field name `PCgenMats`).

(b) The matrices of the minimal generators of the p-group on the standard indecomposable projective module for the algebra (field name `MingenMats`).

(c) The algebra A (field name `Algebra`).

`CompactProjectiveResolutionPGroup(M, n)`

`CompactProjectiveResolution(M, n)`

Computes the projective resolution of the module M out to n steps. The function returns a record with the fields:

(a) The list of the ranks of the projective modules in the resolution (field name `BettiNumbers`).

(b) The record of the boundary maps (field name `ResolutionRecord`).

(c) The module M (field name `Module`).

(d) The augmentation map (field name `AugmentationMap`).

(e) The type of the resolution, whether projective or injective (field name `Typ`).

```
ProjectiveResolutionPGroup(PR)
```

The projective resolution as a complex of modules over the basic algebra of the group algebra, computed from the compact projective resolution PR.

```
ProjectiveResolution(M, n)
```

The projective resolution of the module M computed as a complex out to n steps. The function also returns the augmentation map.

```
ProjectiveResolution(PR)
```

The projective resolution computed from a compact projective resolution PR as a complex. The function also returns the augmentation map.

85.11.3 Cohomology Generators

```
AllCompactChainMaps(PR)
```

Creates the data on the chain maps for all generators of the cohomology of the simple module k in degrees within the limits of the compact projective resolution PR of the simple module. The function returns a record having the following information.

(a) The record of the chain maps of the generators of cohomology (field name `ChainMapRecord`).

(b) The sequence of sizes of the chain map record (field name `ChainSizes`).

(c) The degrees of the chain maps (field name `ChainDegrees`).

(d) The list of cocycles representing the generators (field name `Cocycles`).

(e) The record of the products of the generators (field name `ProductRecord`).

(f) The locations of the products of the generators (field name `ProductLocations`). Much of the information is for use in the computation of the cohomology ring.

```
CohomologyElementToChainMap(P, d, n)
```

Creates a chain map from the projective resolution P to itself for the element number n in degree d of cohomology.

```
CohomologyElementToCompactChainMap(PR, d, n)
```

Creates a chain map in compact form from the compact projective resolution PR to itself for the element number n in degree d of cohomology.

85.11.4 Cohomology Rings

```
CohomologyRing(k, n)
```

```
CohomologyRing(PR, AC)
```

The cohomology ring of the unique simple module k for the basic algebra of the group algebra of a p-group. The input can be given either as the module k and the number of steps n or as the compact projective resolution PR of k together with AC, the calculation of the chain map generators of the cohomology. In the former case the compact resolution and the chain map of the generators are computed in the process. The ring is returned as a record having the following fields:

(a) The polynomial ring or free graded-commutative k-algebra R generated by the cohomology generators (field name `PolRing`).

(b) The ideal of relations in R satisfied by the cohomology generators (field name `RelationsIdeal`).

(c) The list of relations that have been computed (field name `ComputedRelations`).

(d) The chain maps giving the tops of the monomial in the cohomology generators (field name `MonomialData`).

(e) The number of computed steps in the resolution (field name `NumberOfSteps`).

```
MinimalRelations(R)
```

A minimal set of relations generating the relations ideal of a cohomology ring R.

85.11.5 Restrictions and Inflations

```
RestrictionData(A,B)
```

Assuming that A is the basic algebra of a p-group G and that B is the basic algebra of a subgroup of G, the function returns the change of basis matrix that make the standard free module for A into a direct sum of standard free modules for B. It also returns the inverse of the matrix and a set of coset representatives of the `PCGroup(B)` in `PCGroup(A)`.

```
RestrictResolution(PR, RD)
```

Takes the compact projective resolution PR for the trivial module of G and the resolution data RD for the basic algebra of a subgroup H and returns the restriction of the resolution to a complex of modules over the basic algebra for H.

```
RestrictionChainMap(P1,P2)
```

Computes the chain map from the resolution $P2$ of the simple module for the basic algebra of a subgroup H of a group G to the restriction to H of the resolution $P1$ of the simple module for the basic algebra of G. The inputs $P1$ and $P2$ must be in compact form.

```
RestrictionOfGenerators(PR1, PR2, AC1, AC2, REL2)
```

Computes the sequence of images of the generators of the cohomology ring of G restricted to a subgroup H. The input is the projective resolutions and cohomology generators for the basic algebra of G ($PR1$ and $AC1$) and for the basic algebra of the subgroup ($PR2$ and $AC2$), as well as the cohomology relations for the subgroup, $REL2$.

```
InflationMap(PR2, PR1, AC2, AC1, REL1, theta)
```

Returns the images of the generators of the cohomology ring of a quotient group Q in the cohomology ring of a group G. The input θ is the quotient map $G \to Q$. Other input is the projective resolutions and cohomology generators for the basic algebra of G ($PR1$ and $AC1$) and for the quotient group Q ($PR2$ and $AC2$) as well as the cohomology relations for G, $REL1$.

Example H85E21

We create the cohomology ring of a group G of order 64 and find a cyclic subgroup Z of the center of G. We compute the restriction of the cohomology of G to the cohomology of Z and also the inflation of the cohomology of G/Z to the cohomology ring of G.

```
> SetSeed(1);
> G := SmallGroup(64,7);
> Z := sub<G| Random(Center(G))>;
> G;
GrpPC : G of order 64 = 2^6
PC-Relations:
    G.1^2 = G.4,
    G.2^2 = G.5,
    G.3^2 = G.5,
    G.4^2 = G.6,
    G.2^G.1 = G.2 * G.3,
    G.3^G.1 = G.3 * G.5,
    G.3^G.2 = G.3 * G.5
> #Z, [G!Z.i: i in [1 .. Ngens(Z)]];
4 [ G.4 ]
```

So we see that Z has order 4 and is generated by the element $G.4$. Now construct the quotient and the basic Algebras.

```
> Q, mu := quo<G|Z>;
> A := BasicAlgebra(G);
> B := BasicAlgebra(Q);
> C := BasicAlgebra(Z);
```

Next we want the simple modules and the cohomology rings. We compute the cohomology out to 17 steps which should be more than enough to get the generators and relations.

```
> k := SimpleModule(A,1);
> kk := SimpleModule(B,1);
> kkk := SimpleModule(C,1);
> time R := CohomologyRing(k,17);
Time: 2.060
> time S := CohomologyRing(kk,17);
Time: 0.140
> time T := CohomologyRing(kkk,17);
Time: 0.060
```

The structure of the cohomology rings can be read from the following outputs.

```
> R`RelationsIdeal,S`RelationsIdeal,T`RelationsIdeal;
First the cohomology ring for $G$.
Ideal of Graded Polynomial ring of rank 6 over GF(2)
Lexicographical Order
Variables: $.1, $.2, $.3, $.4, $.5, $.6
Variable weights: 1 1 2 2 3 4
Basis:
[
    $.1^2,
    $.1*$.2,
    $.2^3,
    $.1*$.3,
    $.2*$.5,
    $.3^2,
    $.1*$.5 + $.2^2*$.3,
    $.3*$.5,
    $.5^2
]
```

Now the cohomology ring for Q.

```
Ideal of Graded Polynomial ring of rank 4 over GF(2)
Lexicographical Order
Variables: $.1, $.2, $.3, $.4
Variable weights: 1 1 3 4
Basis:
[
    $.1*$.2,
    $.1^3,
    $.1*$.3,
    $.2^2*$.4 + $.3^2
]
```

And finally the cohomology ring for Z.

```
Ideal of Graded Polynomial ring of rank 2 over GF(2)
Lexicographical Order
Variables: $.1, $.2
Variable weights: 1 2
Basis:
[
    $.1^2
]
```

Next we require the inputs for the restriction and inflation maps.

```
> Pr1 := k`CompactProjectiveResolution;
> Pr2 := kk`CompactProjectiveResolution;
> Pr3 := kkk`CompactProjectiveResolution;
> Ac1 := k`AllCompactChainMaps;
> Ac2 := kk`AllCompactChainMaps;
> Ac3 := kkk`AllCompactChainMaps;
```

Now the inflation map from Q to G sends the generators of the cohomology of Q to the given list of elements in the cohomology ring of G.

```
> inf := InflationMap(Pr2,Pr1,Ac2,Ac1,R,mu);
> inf;
[
    $.2,
    $.1,
    $.5,
    $.6
]
```

The restriction map from the cohomology ring of G to the cohomology ring of Z sends the generators of R to the corresponding elements in the computed sequence.

```
> res := RestrictionOfGenerators(Pr1,Pr3,Ac1,Ac3,T);
> res;
[
    0,
    0,
    0,
    $.2,
    0,
    0
]
```

Finally, a set of minimal relations is determined for the cohomology ring R.

```
> MinimalRelations(R);
[
    $.1^2,
    $.1*$.2,
    $.2^3,
    $.1*$.3,
    $.2*$.5,
    $.3^2,
    $.1*$.5 + $.2^2*$.3,
    $.3*$.5,
    $.5^2
```

]

85.12 A-infinity Algebra Structures on Group Cohomology

As described in [Kel01, Kel02], an $A\infty$-algebra structure can be induced on H^*A for any differential graded algebra A. In the case of $\mathrm{Ext}^*_R(S,S)$, for R a quiver algebra quotient and S the direct sum of all simple R-modules, as the homology of the endomorphism algebra of a projective resolution of S, Keller further demonstrates how this additional algebraic structure allows recovery of R from $\mathrm{Ext}^*_R(S,S)$.

An $A\infty$-algebra structure on a vector space V consists of higher structural operations $m_1,\ldots$ defined as $m_i : V^{\otimes i} \to V$ fulfilling the Stasheff axioms for all n:

$$\sum_{i+j-1=n, 0\leq k\leq n-j} \pm m_i(a_1,\ldots,a_{k-1},m_j(a_k,\ldots,a_{k+j}),a_{k+j+1},\ldots,a_n) = 0$$

An $A\infty$-algebra homomorphism from an $A\infty$-algebra A to an $A\infty$-algebra B is a family f_i of maps $A^{\otimes i} \to B$ such that the homomorphism axioms hold for all n:

$$\sum_{i+j-1=n, 0\leq k\leq n-j} \pm f_i(a_1,\ldots,a_{k-1},m_j(a_k,\ldots,a_{k+j}),a_{k+j+1},\ldots,a_n) =$$

$$\sum_{i_1+\ldots+i_r=n} \pm m_r(f_{i_1}(a_1,\ldots,a_{i_1}),\ldots,f_{i_r}(a_{n-i_r+1},\ldots,a_n))$$

According to a theorem fundamental to the algebraic uses of $A\infty$-techniques, for a differential graded algebra A, there is an $A\infty$-structure on H^*A and an $A\infty$-algebra homomorphism $f : H^*A \to A$ such that f_1 is a quasiisomorphism of differential graded algebras, and induced by the identity map on H^*A.

A blackbox method of calculation can be based on Kadeishvilis' proof of this statement, using the homomorphism axioms to recursively calculate any specific values that are needed, and choosing the m_i and f_i in such a way as not to violate the axioms. The following package implements this method for the special case of $\mathrm{Ext}^*_{kG}(k,k)$ for G a p-group, k a prime field of characteristic p and also the one-dimensional unique simple kG-module.

```
AInfinityRecord(G,n)
```

Constructs a record carrying all relevant information to calculate $A\infty$-operations on a group cohomology ring. Among the data carried can be found the cohomology ring in `R`, the cohomology ring quotient in `S`, the projective resolution used in `P`, the simple module resolved in `k` and the basic algebra in `A`.

```
MasseyProduct(Aoo,terms)
HighProduct(Aoo,terms)
```

Given an $A\infty$ object *Aoo* corresponding to a group cohomology ring, this intrinsic calculates the structure map $m_i(t_1 \otimes \ldots \otimes t_i)$, where the i give the length of terms, and $t_1, \ldots, t_i$ are the elements of terms.

```
HighMap(Aoo,terms)
```

Given an $A\infty$ object *Aoo* corresponding to a group cohomology ring, this intrinsic calculates the value of an $A\infty$-quasiisomorphism f at the point $t_1 \otimes \ldots \otimes t_i$, where the i gives the length of terms, and $t_1, \ldots, t_i$ are the elements of terms.

Example H85E22

The $A\infty$-structures on the cohomology rings of cyclic p-groups are well known examples in the literature: the $A\infty$-structure on $H^*(C_n, F_2)$ has one single higher structure nontrivial operation, namely m_n, which takes any n-tuple of odd coclasses to the even coclass of appropriate degree. In order to verify this for a specific example, we start by constructing an $A\infty$ record that contains all relevant information for the cohomology ring.

```
> Aoo := AInfinityRecord(CyclicGroup(4),10);
> S<x,y> := Aoo`S;
> HighProduct(Aoo,[x,x,x,x]);
y
> HighMap(Aoo,[x,x,x,x]);
Basic algebra chain map of degree -1
```

Example H85E23

The code as written handles odd characteristics well, with the sign choices featured in Kadeishvilis article [Kad80] embedded in the code.

```
> Aoo := AInfinityRecord(CyclicGroup(3),10);
> S<x,y> := Aoo`S;
> HighProduct(Aoo,[x,x,x]);
y
> HighMap(Aoo,[x,x,x]);
Basic algebra chain map of degree -1
```

85.12.1 Homological Algebra Toolkit

For the computation of $A\infty$-structures, several methods are used that would invite a wider use in a generic homological algebra toolkit.

```
ActionMatrix(A,x)
```

Produces a matrix of the right action of the Basic algebra element described by x in the Basic algebra A.

```
CohomologyRingQuotient(CR)
```

Computes the actual cohomology ring as a quotient ring of a multivariate polynomial ring from a cohomology ring record.

```
LiftToChainmap(P,f,d)
```

Lifts the function described by f to a chain map from P to P of degree d.

```
NullHomotopy(f)
```

Constructs a null homotopy of the null homotopic chain map f. If f is not null homotopic, the function will throw an error message, since in that case some of the equations encountered on the way are not solvable.

```
IsNullHomotopy(f,H)
```

Confirms that H is a null homotopy of f, in other words that $f = dH - Hd$, with d the differential of the corresponding chain complexes.

```
ChainmapToCohomology(f,CR)
```

Takes a chain map f and returns the element in the cohomology quotient ring to which the chain map corresponds.

```
CohomologyToChainmap(xi,CR,P)
```

Takes an element xi of a cohomology quotient ring of the cohomology ring record CR and a projective resolution corresponding to that cohomology ring, and returns a chain map in the coclass represented by xi.

Example H85E24

To illustrate the code that generates null homotopies, we consider the cohomology ring of a cyclic group, and pick out chain map representatives for the degree 1 coclass. Although this squares to zero, the corresponding chainmaps do not compose to the zero chainmap.

```
> A := BasicAlgebra(CyclicGroup(4));
> k := SimpleModule(A,1);
> P := ProjectiveResolution(k,5);
> R := CohomologyRing(k,5);
> S<x,y> := CohomologyRingQuotient(R);
> xi := CohomologyToChainmap(x,R,P);
> x*x;
```

```
0
> IsZero(xi*xi);
false
> ModuleMaps(xi*xi);
[*
    [0 0 1 0]
    [0 0 0 1]
    [0 0 0 0]
    [0 0 0 0],

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

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

    [0 0 1 0]
    [0 0 0 1]
    [0 0 0 0]
    [0 0 0 0]
*]
> H := NullHomotopy(xi*xi);
> ModuleMaps(H);
[*
    [0 0 0 0]
    [0 0 0 0]
    [0 0 0 0]
    [0 0 0 0],

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

    [0 0 0 0]
    [0 0 0 0]
    [0 0 0 0]
    [0 0 0 0],

    [0 1 0 0]
    [0 0 1 0]
    [0 0 0 1]
    [0 0 0 0],
    [0 0 0 0]
    [0 0 0 0]
    [0 0 0 0]
    [0 0 0 0]
*]
> IsNullHomotopy(xi*xi,H);
true
```

85.13 Bibliography

[**Kad80**] Tornike V. Kadeishvili. On the homology theory of fiber spaces. *Russian Math. Surveys*, (35:3):231–238, 1980. arXiv:math/0504437v1.

[**Kel01**] Bernhard Keller. Introduction to A-infinity algebras and modules. *Homology, Homotopy and Applications*, 3(1):1–35 (electronic), 2001.

[**Kel02**] Bernhard Keller. A-infinity algebras in representation theory. In *Representations of algebra. Vol. I, II*, pages 74–86. Beijing Norm. Univ. Press, Beijing, 2002.

86 QUATERNION ALGEBRAS

Chapter 86
QUATERNION ALGEBRAS

86.1 Introduction

A quaternion algebra A over a field K is a central simple algebra of dimension four over K, or equivalently when K does not have characteristic 2, an algebra generated by elements i, j which satisfy

$$i^2 = a, \ j^2 = b, \ ji = -ij$$

with $a, b \in K^*$. A quaternion algebra in Magma is a specialized type `AlgQuat`, which is a subtype of associative algebra `AlgAss` defined over a field. Magma can recognize if an associative algebra A is a quaternion algebra and will return a standard representation for A with a, b as above.

Examples of quaternion algebras include the ring $M_2(K)$ of 2×2-matrices over K with $a = b = 1$, as well as the division ring of Hamiltonians over $K = \mathbf{R}$ with $a = b = -1$. Every quaternion algebra over a finite field or an algebraically closed field is isomorphic to $M_2(K)$. Over a local field, there is a unique quaternion algebra (up to isomorphism) which is a division ring.

Every quaternion algebra A over K not isomorphic to $M_2(K)$ is a division algebra. Finding a zerodivisor in A is computationally equivalent to the problem of finding a K-rational point on a conic. Given a zerodivisor in A, Magma will compute an explicit isomorphism $A \to M_2(K)$. If L is an extension field of K, then $A_L = A \otimes_K L$ is a quaternion algebra over L, and we say L is a *splitting field* if $A_L \cong M_2(L)$. If $[L : K] = 2$, then L is a splitting field if and only if there exists a K-embedding $L \hookrightarrow A$, and such an embedding can be computed in Magma.

Further functionality is available for quaternion algebras A defined over number fields K. Such an algebra is said to be *unramified* at a noncomplex place v of K if K_v is a splitting field for A, otherwise A is *ramified* at v. Testing if an algebra is ramified at a place is encoded in the *Hilbert symbol*, which can be computed for any such algebra. The set S of ramified places of an algebra is finite and of even cardinality, and conversely, given such a set S there exists a quaternion algebra which is ramified only at S. One may compute the set of ramified places of A as well as constructing an algebra given its ramification set.

In Magma, there are algorithms for quaternion algebras which are analogous to those for number fields—computation of the ring of integers, class group, and unit group. One may compute a maximal order $O \subset A$, a p-maximal order for a prime p of K, and orders with given index in a maximal order. Secondly, a representative set of (right) ideal classes in O can be enumerated, and one can test if two ideals are isomorphic (hence if a given ideal is principal). Finally, for a definite quaternion algebra (defined over a totally real field), one may compute the group of units of norm 1 in O. Over $\mathbf{Q}$, these functions tie into machinery for constructing the Brandt module, with applications to modular forms.

Orders for quaternion algebras over the rational numbers and over rational function fields in MAGMA have type `AlgQuatOrd` and ideals have type `AlgQuatOrdIdl`. Over other number rings, orders have the general type `AlgAssVOrd` and ideals `AlgAssVOrdIdl` which is also the type for used for associative orders. The types `AlgQuatOrd`, `AlgQuatOrdElt` and `AlgQuatOrdIdl` inherit from the types `AlgAssVOrd`, `AlgAssVOrdElt` and `AlgAssVOrdIdl` respectively.

The main reference for material in this chapter is the book of Vignéras [Vig80]. Most nontrivial algorithms for quaternion algebras over the rationals and over number fields are described in [KV10].

IMPORTANT WARNING.

In MAGMA, the rationals are *not* considered to be a number field (the type `FldRat` is not a subtype of `FldNum`). Much of the functionality for quaternion algebras, and particularly for orders in quaternion algebras, is implemented only for algebras whose base field is a `FldNum` (while some, but not all, also works for algebras over the `FldRat`). To compute with algebras over **Q**, in many cases the best solution is to create **Q** as a number field at the outset, using `RationalsAsNumberField()`, and create the algebra over this field instead of `Rationals()`.

86.2 Creation of Quaternion Algebras

A general constructor for a quaternion algebra over any field K creates a model in terms of two generators x and y and three relations

$$x^2 = a, \quad y^2 = b, \quad yx = -xy$$

with $a, b \in K^*$ if K has characteristic different from 2, and

$$x^2 + x = a, \quad y^2 = b, \quad yx = (x+1)y$$

if K has characteristic 2. The printing names i, j, and k are assigned to the generators x, y, and xy by default, unless the user assigns alternatives (see the function `AssignNames` below, or the example which follows).

Special constructors are provided for quaternion algebras over **Q**, which return an algebra with a more general set of three defining quadratic relations. In general, the third generator need not be the product of the first two. This allows the creation of a quaternion algebra A such that the default generators $\{1, i, j, k\}$ form a basis for a maximal order.

```
QuaternionAlgebra< K | a, b >
```

For $a, b \in K^*$, this function creates the quaternion algebra A over the field K on generators i and j with relations $i^2 = a$, $j^2 = b$, and $ji = -ij$ or $ji = (i+1)j$, as $\mathrm{char}(K) \neq 2$ or $\mathrm{char}(K) = 2$, respectively. A third generator is set to $k = ij$. The field K may be of any MAGMA field type; for inexact fields, such as local fields, one should expect unstable results since one cannot test deterministically for element equality.

`AssignNames(~A, S)`

Given a quaternion algebra A and sequence S of strings of length 3, this function assigns the strings to the generators of A, i.e. the basis elements not equal to 1. This function is called by the bracket operators `< >` at the time of creation of a quaternion algebra.

Example H86E1

A general quaternion algebra can be created as follows. Note that the brackets `< >` can be used to give any convenient names to the generators of the ring.

```
> A<x,y,z> := QuaternionAlgebra< RationalField() | -1, -7 >;
> x^2;
-1
> y^2;
-7
> x*y;
z
```

In the case of this constructor the algebra generators are of trace zero and are pairwise anticommuting. This contrasts with some of the special constructors for quaternion algebras over the rationals described below.

Example H86E2

Similarly, we have the following for characteristic 2.

```
> A<i,j> := QuaternionAlgebra< GF(2) | 1, 1 >;
> i^2;
1 + i
> j^2;
1
```

`QuaternionAlgebra(N)`

`Al`	MonStgElt	*Default : "TraceValues"*
`Optimized`	BoolElt	*Default :* `true`

Given a positive squarefree integer N, this function returns a rational quaternion algebra of discriminant N. If the optional parameter `Al` is set to `"Random"`, or N is the product of an even number of primes (i.e. the algebra is indefinite), then a faster, probabilistic algorithm is used. If `Al` is set to `"TraceValues"` and N is a product of an odd number of primes, then an algebra basis is computed which also gives a basis for a maximal order (of discriminant N). If `Optimized` is `true` then an optimized representation of the algebra is returned.

`QuaternionAlgebra(N)`

Given a squarefree polynomial $N \in \mathbf{F}_q[x]$ for some odd q, this function returns a quaternion algebra over $\mathbf{F}_q(x)$ of discriminant N.

`QuaternionAlgebra(I)`

`Optimized` BoolElt *Default :* `true`

Given an ideal I of a number field F with an even number of prime ideal factors, the function returns the quaternion algebra A over F which is ramified exactly at the primes dividing I. The ideal I is not required to be squarefree, so A will be ramified at the radical of I. If `Optimized` is `true` then an optimized representation of the algebra is returned.

`QuaternionAlgebra(I, S)`

`Optimized` BoolElt *Default :* `true`

Given an ideal I and a subset S of real places of a number field F such that the number of prime ideal divisors of I has the same parity as S, the function returns the quaternion algebra which is ramified exactly at the primes dividing I and at the real places specified by the set S. If `Optimized` is `true` then an optimized representation of the algebra is returned.

`QuaternionAlgebra(S)`

`Optimized` BoolElt *Default :* `true`

Given an even set S of noncomplex places of a number field F, this function returns the quaternion algebra which is ramified exactly at S. If `Optimized` is `true` then an optimized representation of the algebra is returned.

Example H86E3

We illustrate the above constructors in the case of a general number field.

```
> P<x> := PolynomialRing(Rationals());
> F<b> := NumberField(x^3-3*x-1);
> Foo := InfinitePlaces(F);
> Z_F := MaximalOrder(F);
> I := ideal<Z_F | 6>;
> A := QuaternionAlgebra(I);
> FactoredDiscriminant(A);
[
    Prime Ideal of Z_F
    Two element generators:
        [3, 0, 0]
        [2, 1, 0],
    Principal Prime Ideal of Z_F
    Generator:
        [2, 0, 0]
]
[]
> A := QuaternionAlgebra(ideal<Z_F | 1>, Foo[1..2]);
> FactoredDiscriminant(A);
[]
[ 1st place at infinity, 2nd place at infinity ]
```

```
QuaternionAlgebra(D1, D2, T)
```

This intrinsic creates the rational quaternion algebra $\mathbf{Q}\langle i, j\rangle$, where $\mathbf{Z}[i]$ and $\mathbf{Z}[j]$ are quadratic suborders of discriminant D_1 and D_2, respectively, and $\mathbf{Z}[ij - ji]$ is a quadratic suborder of discriminant $D_3 = D_1D_2 - T^2$. The values D_1D_2 and T must have the same parity and D_1, D_2 and D_3 must each be the discriminant of some quadratic order, i.e. nonsquare integers congruent to 0 or 1 modulo 4.

Example H86E4

The above constructor is quite powerful for constructing quaternion algebras with given ramification. For any i and j, a commutator element such as $ij - ji$ has trace zero, so in the above constructor, the minimal polynomial of this element is $x^2 + n$, where $n = (D_1D_2 - T^2)/4$.
In the following example we construct such a ring, and demonstrate some of the relations satisfied in this algebra. Note that the minimal polynomial is an element of the commutative polynomial ring over the base field of the algebra.
In particular, we note that the algebra is not in standard form.

```
> A<i,j,k> := QuaternionAlgebra(-7,-47,1);
> PQ<x> := PolynomialRing(RationalField());
> MinimalPolynomial(i);
x^2 - x + 2
> MinimalPolynomial(j);
x^2 - x + 12
> MinimalPolynomial(k);
x^2 - x + 24
> i*j;
k
> i*j eq -j*i;
false
```

From their minimal polynomials, we see that the algebra generators i, j, and k generate commutative subfields of discriminants -7, -47, and -95. The value $82 = (D_1D_2 - T^2)/4$, however, is a more important invariant of the ring. We give a preview of the later material by demonstrating the functionality for computing the determinant and ramified primes of an algebra over $\mathbf{Q}$.

```
> MinimalPolynomial(i*j-j*i);
x^2 + 82
> Discriminant(A);
41
> RamifiedPrimes(A);
```

```
[ 41 ]
```

A ramified prime must be inert or ramified in every subfield and must divide the norm of any commutator element $xy - yx$.

```
> x := i;
> y := j;
> Norm(x*y-y*x);
82
> Factorization(82);
[ <2, 1>, <41, 1> ]
> x := i-k;
> y := i+j+k;
> Norm(x*y-y*x);
1640
> Factorization(1640);
[ <2, 3>, <5, 1>, <41, 1> ]
> KroneckerSymbol(-7,2), KroneckerSymbol(-47,2), KroneckerSymbol(-95,2);
1 1 1
> KroneckerSymbol(-7,41), KroneckerSymbol(-47,41), KroneckerSymbol(-95,41);
-1 -1 -1
```

The fact that the latter Kronecker symbols are -1, indicating that 41 is inert in the quadratic fields of discriminants -7, -47, and -95, proves that 41 is a ramified prime, and 2 is not.

86.3 Creation of Quaternion Orders

Let R be a ring with field of fractions K, and let A be a quaternion algebra over K. An R-*order* in A is a subring $O \subset A$ which is a R-submodule of A with $O \cdot K = A$. An order is *maximal* if it is not properly contained in any other order.

One can create orders for number rings R, for $R = \mathbf{Z}$ or for $R = k[x]$ with k a field. Unlike commutative orders, it is important to note that maximal orders O of quaternion algebras are no longer unique: for any $x \in A$ not in the normalizer of O, we have another maximal order given by $O' = x^{-1}Ox \neq O$.

When $R = \mathbf{Z}$ or $R = k[x]$, the order O has type `AlgQuatOrd`. When R inherits from type `RngOrd` (a number ring), the order O has type `AlgAssVOrd`; see Section 81.4 for more information on constructors and general procedures for these orders.

See above (86.1) for an important warning regarding quaternion algebras over the rationals.

86.3.1 Creation of Orders from Elements

The creation of orders from elements of number rings is covered in Section 81.4. The creation of quaternion orders over the integers and univariate polynomial rings is covered in this section.

```
QuaternionOrder(S)
```

IsBasis BoolElt *Default :* `false`

Given S a sequence of elements in a quaternion algebra defined over $\mathbf{Q}$ or $\mathbf{F}_q(X)$, this function returns the order generated by S over $\mathbf{Z}$ or $\mathbf{F}_q[X]$. If the set S does not generate an order, an error will be returned. If the parameter `IsBasis` is set to `true` then S will be used as the basis of the order returned.

```
QuaternionOrder(R, S)
Order(R, S)
```

Check BoolElt *Default :* `true`

Given a ring R and a sequence S of elements of a quaternion algebra over $\mathbf{Q}$ or $\mathbf{F}_q(X)$, this function returns the R-order with basis S. The sequence must have length four.

Example H86E5

First we construct an order over a polynomial ring.

```
> K<t> := FunctionField(FiniteField(7));
> A<i,j,k> := QuaternionAlgebra< K | t, t^2+t+1 >;
> O := QuaternionOrder( [i,j] );
> Basis(O);
[1, i, j, k ]
```

Next we demonstrate how to construct orders in quaternion algebras generated by a given sequence of elements. When provided with a sequence of elements of a quaternion algebra over $\mathbf{Q}$, Magma reduces the sequence so as to form a basis. When provided with the ring over which these elements are to be interpreted, the sequence must be a basis with initial element 1, and the order having this basis is constructed.

```
> A<i,j,k> := QuaternionAlgebra< RationalField() | -1, -3 >;
> B := [ 1, 1/2 + 1/2*j, i, 1/2*i + 1/2*k ];
> O := QuaternionOrder(B);
> Basis(O);
[ 1, 1/2*i + 1/2*k, 1/2 - 1/2*j, -1/2*i + 1/2*k ]
> S := QuaternionOrder(Integers(),B);
> Basis(S);
[ 1, 1/2 + 1/2*j, i, 1/2*i + 1/2*k ]
```

86.3.2 Creation of Maximal Orders

```
MaximalOrder(A)
```

A maximal order is constructed in the quaternion algebra A. The algebra A must be defined over a field K where K is either a number field, $\mathbf{Q}$, $\mathbf{F}_q(X)$ with q odd, or the field of fractions of a number ring. Over $\mathbf{F}_q(X)$ we use the standard algorithm [Fri97, IR93]. Over the rationals or over a number field, we use a variation of this algorithm optimized for the case of quaternion algebras. First, a factorization of the discriminant of a tame order (see below) is computed. Then, for each prime p dividing the discriminant, a p-maximal order compatible with the existing order is computed. The method used corresponds to Algorithm 4.3.8 in [Voi05]. See also [Voi11].

Example H86E6

The following is an example of a quaternion algebra which is unramified at all finite primes.

```
> P<x> := PolynomialRing(Rationals());
> F<b> := NumberField(x^3-3*x-1);
> A<alpha,beta,alphabeta> := QuaternionAlgebra<F | -3,b>;
> O := MaximalOrder(A);
> Factorization(Discriminant(O));
[]
```

Hence the algebra A has a maximal order of discriminant 1, or equivalently, A is unramified at all finite places of F.

Since we are working over a general order of a number field, we can no longer guarantee that an order will have a free basis, so it must be represented by a pseudomatrix. For more on pseudomatrices, see Section 55.10.

```
> Z_F := BaseRing(O);
> PseudoBasis(O);
[
    <Principal Ideal of Z_F
    Generator:
        Z_F.1, Z_F.1>,
    <Fractional Ideal of Z_F
    Two element generators:
        Z_F.1
        2/3*Z_F.1 + 1/6*Z_F.2 + 1/6*Z_F.3, 3/1*Z_F.1 + i>,
    <Principal Ideal of Z_F
    Generator:
        Z_F.1, j>,
    <Fractional Ideal of Z_F
    Two element generators:
        Z_F.1
        11/2*Z_F.1 + 1/6*Z_F.2 + 35/6*Z_F.3, 3/1*Z_F.1 - i + 3/1*Z_F.1*j + k>
]
```

The wide applicability of the above algorithm, is demonstrated by examining a “random” quaternion algebra over a “random” quadratic number field.

```
> for c := 1 to 10 do
>   D := Random([d : d in [-100..100] | not IsSquare(d)]);
>   K<w> := NumberField(x^2-D);
>   Z_K := MaximalOrder(K);
>   K<K1,w> := FieldOfFractions(Z_K);
>   a := Random([i : i in [-50..50] | i ne 0]) + Random([-50..50])*w;
>   b := Random([i : i in [-50..50] | i ne 0]) + Random([-50..50])*w;
>   printf "D = %o, a = %o, b = %o\n", D, a, b;
>   A := QuaternionAlgebra<K | a,b>;
>   O := MaximalOrder(A);
>   ds := [<pp[1],pp[2],HilbertSymbol(A,pp[1])> :
>            pp in Factorization(Discriminant(O))];
>   print ds;
>   for d in ds do
>     if d[3] eq 1 then
>       break c;
>     end if;
>   end for;
> end for;
D = 5, a = -46/1*K1 + 25/1*w, b = -10/1*K1 - 7/1*w
[
    <Prime Ideal of Z_K
    Two element generators:
        [31, 0]
        [5, 2], 2, -1>,
    <Prime Ideal of Z_K
    Two element generators:
        [11, 0]
        [6, 2], 2, -1>
]
...
```

For each such “random” quaternion algebra, we verify that the Hilbert symbol evaluated at each prime dividing the discriminant of the maximal order is -1, indicating that the algebra is indeed ramified at the prime.

MaximalOrder(O)

For O a quaternion order defined over $\mathbf{Z}$, $\mathbf{F}_q[X]$ with q odd or a number ring, this function returns a maximal order containing O.

```
pMaximalOrder(O, p)
```

For O a quaternion order defined over $\mathbf{Z}$, $\mathbf{F}_q[X]$ with q odd or a number ring and a prime (ideal) p, this function returns a p-maximal order O' containing the order O. The p-adic valuation of the discriminant of O' (which is either 0 or 1) is returned as a second return value.

```
TameOrder(A)
```

Given a quaternion algebra A, this function returns an order O having the property that the odd reduced discriminant of O is squarefree. The algebra A must be defined over a number field or field of fractions of a number ring. The algorithm ignores even primes and does not test the remaining odd primes for maximality.

86.3.3 Creation of Orders with given Discriminant

The following two functions together with the maximal order algorithms of the previous subsection allow the construction of arbitrary Eichler orders.

```
Order(O, N)
```

Given an order O in a quaternion algebra A over the rationals or $\mathbf{F}_q(x)$ with q odd, and some element N in the base ring of O, this function returns a suborder O' of O having index N. Currently, N and the level of O must be coprime and N must have valuation at most 1 at each ramified prime of A. The order O' is locally Eichler at all prime divisors of N that are not ramified in A. In particular, if O is Eichler and N is coprime to the discriminant of A, so is O'.

```
Order(O, N)
```

Given a maximal quaternion order O over a number ring, this function returns an Eichler order of level N inside O.

```
GorensteinClosure(O)
```

Given a quaternion order O over the integers, $\mathbf{F}_q[x]$ with q odd or a number ring, this function returns the smallest Gorenstein order containing O.

Example H86E7

First we construct a quaternion algebra A over $\mathbf{F}_5(x)$ ramified at x^2+x+1, then a maximal order M in A and finally an Eichler O order of discriminant $(x^2+x+1)(x^3+x+1)^5$.

```
> P<x> := PolynomialRing(GF(5));
> A := QuaternionAlgebra(x^2+x+1);
> M := MaximalOrder(A);
> O := Order(M, (x^3+x+1)^5);
> FactoredDiscriminant(O);
[
    <x^2 + x + 1, 1>,
    <x^3 + x + 1, 5>
]
```

86.3.4 Creation of Orders with given Discriminant over the Integers

When constructing quaternion orders over the integers, several shortcuts are available.

```
QuaternionOrder(A, M)
```

Given a quaternion algebra A and a positive integer M, this function returns an order of index M in a maximal order of the quaternion algebra A defined over **Q**. The second argument M can have at most valuation 1 at any ramified prime of A.

```
QuaternionOrder(N)
QuaternionOrder(N, M)
```

Given positive integers N and M, this function returns an order of index M in a maximal order of the rational quaternion algebra A of discriminant N. The discriminant N must be a product of an odd number of distinct primes, and the argument M can be at most of valuation 1 at any prime dividing N. If M is omitted, the integer M defaults to 1, i.e., the function will return a maximal order.

```
QuaternionOrder(D1, D2, T)
```

This intrinsic constructs the quaternion order $\mathbf{Z}\langle x, y\rangle$, where $\mathbf{Z}[x]$ and $\mathbf{Z}[y]$ are quadratic subrings of discriminant D_1 and D_2, respectively, and $\mathbf{Z}[xy - yx]$ is a quadratic subring of discriminant $D_1 D_2 - T^2$.

Note that the container algebra of such a quaternion order is **not** usually in standard form (see the example below).

Example H86E8

The above constructors permit the construction of Eichler orders over **Z**, if the discriminant N and the index M are coprime. More generally they allow the construction of an order whose index in an Eichler order divides the discriminant.

```
> A := QuaternionOrder(103,2);
> Discriminant(A);
206
> Factorization($1);
[ <2, 1>, <103, 1> ]
> _<x> := PolynomialRing(Rationals());
> [MinimalPolynomial(A.i) : i in [1..4]];
[
    x - 1,
    x^2 + 1,
    x^2 - x + 52,
    x^2 + 104
]
```

The constructor `QuaternionOrder(D1, D2, T)` may return an order whose container algebra is not in standard form.

```
> A := QuaternionOrder(-4, 5, 2);
> B := Algebra(A);
> B.1 * B.2 eq - B.2 * B.1;
false
```

86.4 Elements of Quaternion Algebras

For more information about elements of orders of associative algebras, see Section 81.4.

86.4.1 Creation of Elements

`A ! 0`

`Zero(A)`

The zero element of the quaternion algebra A.

`A ! 1`

`One(A)`

The identity element of the quaternion algebra A.

`A . i`

`Name(A, i)`

Given a quaternion algebra A and an integer $1 \leq i \leq 3$, returns the ith generator of A as an algebra over the base ring. Note that the element 1 is always the first element of a basis, and is never returned as a generating element.

`A ! x`

Return an element of the quaternion algebra A described by x, where x may be an algebra element, a module element, a sequence, an element of an order of an associative algebra or be coercible into the coefficient ring of A.

86.4.2 Arithmetic of Elements

`x + y`

The sum of x and y.

`x - y`

The difference of x and y.

`x * y`

The product of x and y.

`x / y`

The quotient of x by the unit y in the quaternion algebra.

```
x eq y
```

Returns `true` if the elements x and y are equal; otherwise `false`.

```
x ne y
```

Returns `true` if and only if the elements x and y are not equal.

```
x in A
```

Returns `true` if and only if x is in the algebra A.

```
x notin A
```

Returns `true` if and only if x is not in the algebra A.

```
Conjugate(x)
```

The conjugate $\bar{x}$ of the element x of a quaternion algebra, defined so that the reduced trace and reduced norm are $\bar{x} + x$ and $\bar{x}x$, respectively.

```
ElementToSequence(x)
Eltseq(x)
Coordinates(x)
```

Given an element x of a quaternion algebra or order, this function returns the sequence of coordinates of x in terms of the basis of its parent.

```
Norm(x)
```

The reduced norm $\mathrm{N}(x)$ of the element x of a quaternion algebra, defined so that the characteristic polynomial for x is $x^2 - \mathrm{Tr}(x)x + \mathrm{N}(x) = 0$, where $\mathrm{Tr}(x)$ is the reduced trace.

```
Trace(x)
```

The reduced trace $\mathrm{Tr}(x)$ of the element x of a quaternion algebra, defined so that the characteristic polynomial for x is $x^2 - \mathrm{Tr}(x)x + \mathrm{N}(x) = 0$, where $\mathrm{N}(x)$ is the reduced norm.

```
CharacteristicPolynomial(x)
```

The characteristic polynomial of degree 2 for the element x of a quaternion algebra over the base ring of its parent.

```
MinimalPolynomial(x)
```

The minimal polynomial of degree 1 or 2 for the element x of a quaternion algebra over the base ring of its parent.

Example H86E9

We demonstrate the relation between characteristic polynomial, and reduced trace and norm in the following example.

```
> A := QuaternionAlgebra< RationalField() | -17, -271 >;
> x := A![1,-2,3,0];
> Trace(x);
2
> Norm(x);
2508
> x^2 - Trace(x)*x + Norm(x);
0
```

Note that trace and norm of an element x of any algebra can be defined as the trace and norm of the linear operator corresponding to right-multiplication by x. The reduced trace and norm in a quaternion algebra A are taken instead to be the corresponding trace and determinant in any two-dimensional matrix representation of A, or equivalently, the sum and product of an element with its conjugate. The definition of norm and trace used for a general algebra can be realised in a quaternion algebra by the following code.

```
> P<X> := PolynomialRing(RationalField());
> M := RepresentationMatrix(x, A);
> M;
[   1   -2    3    0]
[  34    1    0    3]
[-813    0    1    2]
[   0 -813  -34    1]
> Trace(M);
4
> Factorization(CharacteristicPolynomial(M));
[
    <X^2 - 2*X + 2508, 2>
]
```

The general definition of trace (for the algebra) is twice the reduced trace, and the general definition of norm is the square of the reduced norm.

86.5 Attributes of Quaternion Algebras

`BaseField(A)`

`BaseRing(A)`

The base field of the quaternion algebra A.

`Basis(A)`

The basis of the algebra A.

`RamifiedPrimes(A)`

Given a quaternion algebra A over $\mathbf{Q}$ or $\mathbf{F}_q(X)$ with q odd, this function returns a list of primes or normalized irreducible polynomials corresponding to the finite ramified places of A.

Example H86E10

The sequence of ramified primes of a quaternion algebra A over $\mathbf{Q}$ determines the isomorphism class of the algebra.

```
> A := QuaternionAlgebra(-436,-503,22);
> RamifiedPrimes(A);
[ 17 ]
```

Provided the discriminant is of a size which can be factored, the ramified primes are determined efficiently using Hilbert symbols.

`RamifiedPlaces(A)`

`FactoredDiscriminant(A)`

Given a quaternion algebra A over $\mathbf{Q}$ or $\mathbf{F}_q(X)$ with q odd or a number field, this function returns the finite as well as infinite places where A is ramified.

Note: The first return value of these functions is always a list of ideals, even if the algebra is given over $\mathbf{Q}$ or $\mathbf{F}_q(X)$.

Example H86E11

This example shows the (minor) difference between `RamifiedPrimes` and `RamifiedPlaces`.

```
> F<x> := RationalFunctionField( GF(5) );
> A := QuaternionAlgebra< F | 2, x >;
> R<x>:= Integers(F);
> RamifiedPrimes(A);
[ x ]
> RamifiedPlaces(A);
[
    Ideal of Univariate Polynomial Ring in x over GF(5) generated by x
]
[ Infinity ]
```

`Discriminant(A)`

The reduced discriminant of a quaternion algebra A over $\mathbf{Q}$, $\mathbf{F}_q(X)$ with q odd or a number field. In the first two cases, the functions return the product of the ramified primes. Over number fields, they return the product of the ramified prime ideals as well as the sequence of ramified infinite places.

`StandardForm(A)`

Returns integers a and b in the base field F of the given quaternion algebra A such that there exists elements $i, j \in A$ where $i^2 = a$, $j^2 = b$, and $ji = -ij$. The third object returned is the standard quaternion algebra $B =$ `QuaternionAlgebra<F|a,b>`, and the fourth object is the homomorphism from A to B.

86.6 Hilbert Symbols and Embeddings

Let A be a quaternion algebra over Q, $\mathbf{F}_q(X)$ (with q odd) or a number field F with defining elements a, b, and let v be a place of F. If v is unramified in A (i.e. $A \otimes_F F_v \cong M_2(F_v)$, we define the *Hilbert symbol* $(a, b)_v$ to be 1, and otherwise we define $(a, b)_v = -1$.

`HilbertSymbol(a, b, p)`

`HilbertSymbol(A, p)`

`Al` MonStgElt *Default : "NormResidueSymbol"*

Computes the Hilbert symbol for the quaternion algebra A over F, namely $(a, b)_p$, where $a, b \in F$ and p is either a prime (if $a, b \in \mathbf{Q}$ or $\mathbf{F}_q(X)$) or a prime ideal. If $a, b \in \mathbf{Q}$, by default table-lookup is used to compute the Hilbert symbol; one can optionally insist on using the full algorithm by setting the parameter `Al` to the value `"Evaluate"`.

`IsRamified(p, A)`

`IsUnramified(p, A)`

Returns `true` if and only if the prime or prime ideal p is ramified (unramified) in the quaternion algebra A.

Example H86E12

We first verify the correctness of all Hilbert symbols over the rationals.

```
> QQ := Rationals();
> for a,b in [1..8] do
>   bl := HilbertSymbol(QQ ! a, QQ ! b,2 : Al := "Evaluate")
>               eq NormResidueSymbol(a,b,2);
>   print <a,b,bl>;
>   if not bl then
>     break a;
>   end if;
> end for;
<1, 1, true>
<1, 2, true>
<1, 3, true>
...
```

For a second test, we input a quaternion algebra which is unramified at all finite places.

```
> P<x> := PolynomialRing(Rationals());
> F<b> := NumberField(x^3-3*x-1);
> Z_F := MaximalOrder(F);
> A := QuaternionAlgebra<F | -3,b>;
> symbols := [];
> for p in [p : p in [2..100] | IsPrime(p)] do
>   pps := Decomposition(Z_F,p);
>   for pp in pps do
>     Append(~symbols,HilbertSymbol(A,pp[1]));
>   end for;
> end for;
> symbols;
[ 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1 ]
```

Finally, we test "random" quaternion algebras over quadratic extensions at even primes, the hardest case. We use the fact that the quaternion algebra (a, b) is ramified at a prime ideal p if and only if b is a norm from the extension $F(\sqrt{a})$, so we can test this condition using IsLocalNorm. Note that this takes substantially more time.

```
> for c in [2,-2,6,-6,-1,3,-3] do
>   K<s> := NumberField(x^2-c);
>   Z_K := MaximalOrder(K);
>   Z_Kmod8, f8 := quo<Z_K | 8>;
>   PPK<xK> := PolynomialRing(K);
>   for i := 1 to 10 do
>     S := [x+y*Z_K.2 : x,y in [0..7] | x*y ne 0];
>     a := Random(S);
>     b := Random(S);
>     A := QuaternionAlgebra<K | a,b>;
>     for pp in Decomposition(Z_K,2) do
>       hsym := HilbertSymbol(A,pp[1]);
>       if not IsIrreducible(xK^2-a) then
>         print <c, a, b, hsym eq 1>;
>         if hsym ne 1 then
>           break c;
>         end if;
>       else
>         lclsym := IsLocalNorm(AbelianExtension(ext<K | xK^2-a>),Z_K ! b,pp[1]);
>         bl := (hsym eq 1) eq lclsym;
>         print <c, a, b, bl>;
>         if not bl then
>           break c;
>         end if;
>       end if;
>     end for;
>   end for;
> end for;
<2, 5/1*Z_K.1 + 3/1*Z_K.2, Z_K.1 + 7/1*Z_K.2, true>
<2, 6/1*Z_K.1 + 4/1*Z_K.2, 4/1*Z_K.1 + Z_K.2, true>
```

```
<2, 7/1*Z_K.1 + Z_K.2, 2/1*Z_K.1 + 2/1*Z_K.2, true>
...
```

```
pMatrixRing(A, p)
```

```
pMatrixRing(O, p)
```

Precision RngIntElt *Default :*

Let A be a quaternion algebra A over a field F where F is the rationals, a number field of $\mathbf{F}_q(x)$ with q odd. Given A and a prime (ideal) p of the ring of integers R of F such that p is unramified in A, this function returns the matrix ring over the completion F_p of F at p, a map from $A \to M_2(F_p)$ and the embedding $F \to F_p$.

Given a p-maximal order O in A, the map from $A \to M_2(F_p)$ induces a map from $O \to \mathrm{M}_2(R_p)$.

```
IsSplittingField(K, A)
```

```
HasEmbedding(K, A)
```

ComputeEmbedding BoolElt *Default :* false

Given a quaternion algebra A defined over $\mathbf{Q}$, $\mathbf{F}_q(X)$ (with q odd) or a number field F and K a quadratic extension of F, the function returns true if and only if there exists an embedding $K \to A$ over F. This is done by comparison of ramified places in K and A (see [Vig80, Cor. III.3.5]). If no embedding exists, the second return value will be a witness place. If an embedding exists and the optional argument ComputeEmbedding is set to true, the second and third return values contain the result of a call to Embed as described below.

```
Embed(K, A)
```

Al MonStgElt *Default :* "*NormEquation*"

Given a quaternion algebra A defined over $\mathbf{Q}$, $\mathbf{F}_q(X)$ (with q odd) or a number field F and K a quadratic extension of F, returns an embedding $K \to A$ over F, given as an element of A, the image of the primitive generator of K, and the map $K \to A$.

The algorithm by default involves solving a relative norm equation. Alternatively, a naive search algorithm may be selected by setting the optional parameter Al:="Search".

If there is no embedding, a runtime error occurs (or the "Search" runs forever). To check whether an embedding exists, use HasEmbedding (see immediately above).

```
Embed(Oc, O)
```

Al MonStgElt *Default : "NormEquation"*

Given a quadratic order O_c with base number ring R and a quaternion order O with base ring R, the function computes an embedding $O_c \hookrightarrow O$ over R. It returns the image of the second generator `Oc.2` of `Oc`; secondly it returns the embedding map $O_c \to O$.

The algorithm by default involves solving a relative norm equation. Alternatively, a naive search algorithm may be selected by setting the optional parameter `Al:="Search"`.

Notes. Let K be the number field containing Oc.

(i) `Oc.1, Oc.2` are the generators of Oc as a module, and `Oc.2` is unrelated to `K.1`, where K is the number field containing Oc.

(ii) To check whether an embedding of K into the algebra exists, one can use `HasEmbedding(K, Algebra(O) : ComputeEmbedding:=false)`.

Example H86E13

```
> F<b> := NumberField(Polynomial([1,-3,0,1]));
> A := QuaternionAlgebra<F | -3, b>;
> K := ext<F | Polynomial([2,-1,1])>;
> mu, iota := Embed(K, A);
> mu;
1/2 + 1/6*(-2*b^2 + 2*b + 7)*i + 1/2*(2*b^2 + b - 6)*j + 1/6*(-2*b^2 - b + 4)*k
> MinimalPolynomial(mu);
$.1^2 - $.1 + 2
> iota(K.1) eq mu;
true
```

86.7 Predicates on Algebras

A quaternion algebra A over a number field F with $[F : \mathbf{Q}] = h$ is *definite* (or *totally definite*) if F is totally real and $A \otimes_{\mathbf{Q}} \mathbf{R} \cong H^h$, where H is the division ring of real Hamiltonians, otherwise A is *indefinite.*

A quaternion algebra A over $\mathbf{F}_q(X)$ is called *definite* if the place corresponding to the degree valuation is ramified.

```
IsDefinite(A)
```

```
IsIndefinite(A)
```

Given a quaternion algebra A over a number field, $\mathbf{Q}$ or $\mathbf{F}_q(X)$ with q odd, returns `true` if and only if A is a (totally) definite or indefinite quaternion algebra, respectively.

86.8 Recognition Functions

A quaternion algebra A over a field K is isomorphic to the matrix ring $M_2(K)$ if and only if there exists a zerodivisor ϵ in A. Given such an ϵ, we can exhibit an explicit isomorphism; otherwise a zerodivisor will be computed first by finding a point on a conic (see [Vig80, Cor. I.2.4]).

Given an associative algebra, we also have an algorithm to recognize if the algebra is a quaternion algebra, and, if so, return an isomorphism to a quaternion algebra in standard form.

```
IsMatrixRing(A)
```

`Isomorphism` BoolElt *Default :* `false`

Returns `true` if and only if the quaternion algebra A with base field F is isomorphic to $M_2(F)$, or equivalently if A has no ramified places. The field F has to be $\mathbf{Q}$, $\mathbf{F}_q(X)$ (with q odd) or a number field.

If A is isomorphic to $M_2(F)$ and `Isomorphism` is set to `true`, then $M_2(F)$ and an isomorphism $A \to M_2(F)$ are also returned.

```
MatrixRing(A, eps)
```

```
MatrixAlgebra(A, eps)
```

Given a quaternion algebra A and a zerodivisor $\epsilon \in A$, the function returns the matrix algebra $M_2(F)$ and an isomorphism $A \to M_2(F)$.

Example H86E14

```
> A := QuaternionAlgebra<Rationals() | -1, 1>;
> eps := A.3-1;
> MinimalPolynomial(eps), Norm(eps);
x^2 + 2*x
0
```

Thus, since ϵ has reduced norm 0, it is a zerodivisor: indeed, $\epsilon(\epsilon + 2) = 0$.

```
> M2F, phi := MatrixRing(A,eps);
> [<MinimalPolynomial(A.i), MinimalPolynomial(phi(A.i))> : i in [1..3]];
[
    <x^2 + 1, x^2 + 1>,
    <x^2 - 1, x^2 - 1>,
    <x^2 - 1, x^2 - 1>
]
```

```
IsQuaternionAlgebra(B)
```

Returns `true` if and only if the associative algebra B is a quaternion algebra; if true, it returns the associated quaternion algebra A in standard form and an algebra homomorphism from B to A. The algorithm used is [Voi05, Algorithm 4.2.9].

Example H86E15

We create an associative algebra which is known to be a quaternion algebra A and then recover A (or an isomorphic algebra).

```
> A := AssociativeAlgebra(QuaternionAlgebra<Rationals() | -1,1>);
> vecs := [&+[Random(10)*A.i : i in [1..4]] : j in [1..4]];
> Mchange := Matrix(Rationals(),4,4,&cat[Eltseq(vecs[i]) : i in [1..4]]);
> Mchange := Mchange^(-1);
> seq := [<i,j,k,((vecs[i]*vecs[j])*Mchange)[k]> : i,j,k in [1..4]];
> A := AssociativeAlgebra<Rationals(),4 | seq>;
> bl, Aquat, phi := IsQuaternionAlgebra(A);
> bl;
true
> Aquat;
Quaternion Algebra with base ring Rational Field
> Aquat.1^2, Aquat.2^2;
25 -3924/25
> phi;
Mapping from: AlgAss: A to AlgQuat: Aquat given by a rule
```

We now verify the functionality when a zerodivisor is encountered.

```
> A := Algebra(MatrixAlgebra(Rationals(),2));
> IsQuaternionAlgebra(A);
true Quaternion Algebra with base ring Rational Field
Mapping from: AlgAss: A to Quaternion Algebra with base ring Rational Field
given by a rule
```

The algebra $k < x, y >$ with $x^2 = y^2 = xy + yx = 0$ is not semisimple; the ideal generated by x, y is a nontrivial two-sided ideal. Similarly, a commutative algebra is not a quaternion algebra.

```
> A := Algebra(FPAlgebra<Rationals(), x,y | x^2, y^2, x*y+y*x>);
> IsQuaternionAlgebra(A);
false
> A := Algebra(FPAlgebra<Rationals(), x | x^4+x^2+1>);
> IsQuaternionAlgebra(A);
false
```

In characteristic 2, the algorithm also performs correctly, both for an associative but non-quaternion algebra and for the "universal" example of a quaternion algebra.

```
> A := Algebra(FPAlgebra<GF(2), x,y | x^2, y^2, x*y+y*x+1>);
> IsQuaternionAlgebra(A);
false
> F<a,b,x,y,z,w> := FieldOfFractions(PolynomialRing(GF(2),6));
> M := [[1,0,0,0],[0,1,0,0],[0,0,1,0],[0,0,0,1],
>       [0,1,0,0],[a,1,0,0],[0,0,0,1],[0,0,a,1],
>       [0,0,1,0],[0,0,1,1],[b,0,0,0],[b,b,0,0],
>       [0,0,0,1],[0,0,a,0],[0,b,0,0],[a*b,0,0,0]];
> A<alpha,beta> := AssociativeAlgebra<F,4 | M>;
> alpha^2+alpha+a;
(0 0 0 0)
> beta^2+b;
(0 0 0 0)
>
> bl, Aquat, phi := IsQuaternionAlgebra(A);
> bl;
true
> Aquat;
Quaternion Algebra with base ring Multivariate rational function field of
rank 6 over GF(2)
> theta := phi(x+y*alpha+z*beta+w*alpha*beta);
> Trace(theta);
y
> Norm(theta);
a*b*w^2 + a*y^2 + b*z^2 + b*z*w + x^2 + x*y
```

`MatrixRepresentation(A)`

`MatrixRepresentation(R)`

Given a quaternion algebra A over $\mathbf{Q}$ or a quaternion order R over $\mathbf{Z}$, this function returns a 2×2-matrix representation of A, defined over a quadratic extension.

86.9 Attributes of Orders

For further information about orders of associative algebras, see Section 81.4.

For a quaternion order S over $\mathbf{Z}$ or $\mathbf{F}_q[X]$, Magma additionally defines the following functions.

`Algebra(S)`

`QuaternionAlgebra(S)`

The quaternion algebra for which S is an order.

`BasisMatrix(S)`

`EmbeddingMatrix(S)`

Returns the basis matrix of the quaternion order S over $\mathbf{Z}$ or $\mathbf{F}_q[X]$. The rows of the matrix give the basis elements of S with respect to the basis of the container algebra.

`Discriminant(S)`

Given an order S over $\mathbf{Z}$ or $\mathbf{F}_q[X]$, this function returns the reduced discriminant of S as a positive integer or a normalized polynomial.

`FactoredDiscriminant(S)`

Given a quaternion order S, this function returns the factorisation of the reduced discriminant of S (that is, `Factorization(Discriminant(S))`).

`Conductor(S)`

`Level(S)`

Given an order S over $\mathbf{Z}$ or $\mathbf{F}_q[X]$ in a quaternion algebra A, this function returns the reduced index of S in a maximal order of A containing it. Together with the reduced discriminant of the order, this serves to classify the local isomorphism class of an Eichler order.

`Normalizer(S)`

Let S be an order in a definite quaternion algebra A over a field F where F is the rationals, $\mathbf{F}_q(t)$ or a number field. This function returns a matrix group G isomorphic to the normalizer of S in A^* modulo F^*. A homomorphism from G to A^* is also returned.

86.10 Predicates of Orders

Let O be a quaternion order with base ring $\mathbf{Z}$, $\mathbf{F}_q[X]$ with q odd, or a number ring. Then Magma can test the following predicates.

`IsMaximal(O)`

Returns `true` if and only if the order O is maximal.

`IspMaximal(O, p)`

Returns `true` if and only if the order O is maximal at the prime or prime ideal p.

`IsEichler(O)`

`MaximalOrders` BoolElt *Default :* `false`

Returns `true` if and only if the order O is *Eichler*, that is an intersection of two (not necessarily distinct) maximal orders. The function calls the `EichlerInvariant` intrinsic explained below.

If the optional argument `MaximalOrders` is set to `true`, the algorithm also returns two maximal orders such that O is their intersection.

`IsEichler(O, p)`

`MaximalOrders` BoolElt *Default :* `false`

Returns `true` if and only if the completion of the order O at the prime (ideal) p is *Eichler*.

If the optional argument `MaximalOrders` is set to `true`, the algorithm also returns two p-maximal orders such that O is their intersection.

```
EichlerInvariant(O, p)
```

Returns the local Eichler invariant of O at some prime (ideal) p which divides the discriminant of O. Let R be the base ring of O and let J be the Jacobson radical of the R/p-algebra O/pO. If J has dimension 3 then the Eichler invariant is defined to be 0. Otherwise the quotient of O/pO by J is either isomorphic to a direct sum of two copies of R/p or a quadratic field extension of R/p. In the first case the Eichler invariant is 1, in the latter it is -1.

```
IsHereditary(O)
```

Returns `true` if and only if the order O is a hereditary order in a quaternion algebra A. That is, every lattice in A of full rank such that O is contained in its left order is a projective left O-module. The hereditary orders are precisely those with squarefree discriminant.

```
IsHereditary(O, p)
```

Returns `true` if and only if the completion of the order O at the prime (ideal) p is *hereditary*.

```
IsGorenstein(O)
```

Returns `true` if and only if the order O is a Gorenstein order. That is, the dual of O with respect to the trace bilinear form is a projective O-module.

```
IsGorenstein(O, p)
```

Returns `true` if and only if the completion of the order O at the prime (ideal) p is *Gorenstein*.

86.11 Operations with Orders

```
O1 meet O2
```

This returns the order obtained by intersecting the quaternion orders O_1 and O_2.

```
O ^ x
```

This returns the conjugate of the order O by an element x in the associative algebra A for which O is an order (in other words the order $x^{-1}Ox$).

86.12 Ideal Theory of Orders

The right (or left) ideals of an R-order O in a quaternion algebra A defined over a number field fall into finitely many isomorphism classes. Already, for the case of number rings, it is a computationally difficult problem to compute the class group, and due to the noncommutativity of quaternion algebras, the problem of enumerating ideal classes of quaternion orders is even more difficult because this set does not have the structure of a group.

All orders in this section are required to be *Eichler*. The reader should recall that this includes the maximal orders.

The algorithms underpinning the intrinsics described in this section, in particular, those for determining ideal classes, are described in [KV10]. The methods depend on whether A is definite or indefinite. For a definite algebra, we use a variation of Kneser's neighbouring method together with Eichler's mass formula (see [Vig80, Chapter 5] and [DG88]) to construct sufficient ideals to represent all ideal classes. We then test if two right (or left) ideals I, J are isomorphic (Section 86.14.3).

For an indefinite quaternion algebra, we use a theorem of Eichler [Rei03, Th. 35.14] which states that the reduced norm gives rise to an bijection of sets between the class groups of the order and the number ring. (Over $\mathbf{Z}$ or $\mathbf{F}_q[X]$, therefore, every ideal of an indefinite quaternion order is principal.) We may compute representative ideals I of O whose reduced norm is in a specified ideal class. We then reduce the problem of testing for an isomorphism $I \cong J$ between two ideals to the similar problem over R. Here, the problem of explicitly computing such an isomorphism is much less difficult, as an order (or ideal) will have infinitely many elements of bounded norm, and in most cases a search amongst reduced bases will find a desired element.

Ideals of quaternion orders belong to the types `AlgQuatOrdIdl` and `AlgAssVOrdIdl` according as to whether the order O is defined over $\mathbf{Z}$ or $\mathbf{F}_q[X]$ (both of type `AlgQuatOrd`) or over a number ring (of type `AlgAssVOrd`). For more on the constructions and functions for ideals of the latter type, see Section 81.4.

86.12.1 Creation and Access Functions

```
LeftIdeal(S, X)
```

```
lideal< S | X >
```

```
RightIdeal(S, X)
```

```
rideal< S | X >
```

```
ideal< S | X >
```

These intrinsics construct a left, right or two-sided ideal of the order S generated by the sequence X, which should contain elements that are coercible into the algebra of S.

In the case of the constructors `lideal< | >` and `rideal< | >`, the right hand side may also be a matrix or pseudo-matrix giving the basis of the ideal with respect to the basis of S.

`PrimeIdeal(S, p)`

Given a quaternion order S over $\mathbf{Z}$ or $\mathbf{F}_q[X]$, the function returns the unique two-sided (integral) prime ideal P of S over the prime p of $\mathbf{Z}$ or $\mathbf{F}_q[X]$. If p exactly divides the discriminant of S, then P properly contains pS and need not be principal, and otherwise P is equal to pS.

`CommutatorIdeal(S)`

The two-sided ideal of the quaternion order S generated by elements of the form $xy - yx$.

`MaximalLeftIdeals(O, p)`

`MaximalRightIdeals(O, p)`

The integral ideals of norm p with left or right order O.

Example H86E16

We demonstrate the construction of the 2-sided prime ideals and their relationship with the commutator ideal.

```
> S := QuaternionOrder(2*5*11);
> P := PrimeIdeal(S, 2);
> I := ideal< S | [2] cat [ x*y-y*x : x, y in Basis(S) ] >;
> P eq I;
true
> Q := PrimeIdeal(S, 5);
> R := PrimeIdeal(S, 11);
> P*Q*R eq CommutatorIdeal(S);
true
```

By way of explanation, we note that the composition of ideals is well-defined, since each of these ideals is a 2-sided ideal for S, that is, a left ideal whose right order is also S. The collection of all prime ideals over the ramified primes of a maximal order forms an elementary 2-abelian class group, and the commutator ideal is the product of these prime ideals.

Example H86E17

First we create an integral ideal I of norm 2.

```
> QQ:= Rationals();
> A<i,j> := QuaternionAlgebra< QQ | -1, -11 >;
> S := MaximalOrder(A);
> P<x> := PolynomialRing(QQ);
> P ! MinimalPolynomial(i);
x^2 + 1
> I := lideal< S | 2, 1+i >;
> Norm(I);
2
> I in MaximalLeftIdeals(S, 2);
```

```
true
```

We now examine the basis of the ideal I.

```
> Basis(I);
[ 2, 1 + i, i + k, 3/2 + 1/2*i + 1/2*j + 1/2*k ]
> [ Eltseq(x) : x in Basis(I) ];
[
    [ 2, 0, 0, 0 ],
    [ 1, 1, 0, 0 ],
    [ 0, 1, 0, 1 ],
    [ 3/2, 1/2, 1/2, 1/2 ]
]
> BasisMatrix(I, A);
[  2   0   0   0]
[  1   1   0   0]
[  0   1   0   1]
[3/2 1/2 1/2 1/2]
```

If the ideal I is printed, the rational coordinates with respect to the basis of the quaternion order S will be shown, We can get this base change matrix if we call `BasisMatrix` with no additional parameters.

```
> I;
Left Ideal with basis Pseudo-matrix over Integer Ring
[2 0 0 0]
[1 1 0 0]
[0 0 2 0]
[1 0 1 1]
> BasisMatrix(I);
[2 0 0 0]
[1 1 0 0]
[0 0 2 0]
[1 0 1 1]
```

`LeftOrder(I)`

Given an ideal I of a quaternion order defined over $\mathbf{Z}$, $\mathbf{F}_q[X]$ or a number ring, this function returns the left order of I, defined as the ring of all elements of the quaternion algebra of I mapping I to itself under left multiplication.

`RightOrder(I)`

Given an ideal I of a quaternion order defined over $\mathbf{Z}$, $\mathbf{F}_q[X]$ or a number ring, this function returns the right order of I, defined as the ring of all elements of the quaternion algebra of I mapping I to itself under right multiplication.

Example H86E18

```
> K := NumberField(Polynomial([5,0,1]));
> K;
Number Field with defining polynomial $.1^2 + 5 over the Rational Field
> A := QuaternionAlgebra<K | 3, K.1>;
> O := MaximalOrder(A);
> I := lideal<O | O.2, Norm(O.2)>;
> Norm(I);
Principal Ideal
Generator:
    2/1*$.1
> LeftOrder(I) eq O;
true
> RightOrder(I) eq O;
true
```

86.12.2 Enumeration of Ideal Classes

The tools provided for calculating ideal classes in the case of a maximal or, more generally, an Eichler order S in a quaternion algebra over $\mathbf{Q}$, $\mathbf{F}_q(X)$ (with q odd) or a number field are described in this section.

`Mass(S)`

Given a definite order S of level N over R, this function returns the mass

$$\sum_i h_i/[S_i^* : R^*]$$

where $S_1, \ldots, S_t$ represent the conjugacy classes of Eichler orders of level N and h_i denotes the number of two-sided ideal classes of S_i.

`LeftIdealClasses(S)`

`RightIdealClasses(S)`

`Support` [RngOrdIdl] *Default :*

These intrinsics find representatives for the left or right locally free ideal classes of S, where S is an order in a quaternion algebra A defined over $\mathbf{Q}$, $\mathbf{F}_q(X)$ (with q odd) or a number field. The algorithms are guaranteed to work correctly only when S is a maximal order or an Eichler order.

For definite algebras, the support of the returned ideals may be specified using parameter `Support`; this should be a sequence of primes or prime ideals in the base ring which generate the narrow class group and which do not ramify in A. In this case, the routine will find ideal class representatives whose norms are divisible only by primes in the specified support. The algorithm usually runs faster if the support is either not specified or if the support includes the prime divisors of the discriminant of S.

For indefinite algebras, the representatives are obtained from a ray class group of the base field.

`TwoSidedIdealClasses(S)`

`Support` [RngOrdIdl] *Default :*

Given an order S in a quaternion algebra, this function returns a sequence containing representatives for the two-sided locally free ideal classes of S. The algorithm is only guaranteed to work when S is a maximal order or an Eichler order.

If the optional argument `Support` is specified, Magma tries to construct representatives that have norm divisible by primes or prime ideals in the specified support. If this is not possible, the set specified using parameter `Support` will be enlarged by the prime divisors of the discriminant of S. If this enlarged set is still not large enough (which can only happen if the set does not generate the class of group of the base ring of S) an error will be raised.

`TwoSidedIdealClassGroup(S : Support)`

`Support` [RngOrdIdl] *Default :*

Given an order S in a quaternion algebra, this returns the two-sided ideal class group of S as an abstract abelian group, together with a map from the group to the set of two-sided ideal classes of S. Inverses with respect to this map can be calculated in a very efficient way and thus may be used to compute discrete logarithms. The algorithm is only guaranteed to work when S is a maximal order or an Eichler order.

The optional parameter `Support` may be used to specify a support: this should be a sequence of primes or prime ideals in the base ring which generate the narrow class group and which do not ramify in the parent algebra.

`ConjugacyClasses(S)`

Given a maximal order (or an Eichler order S of level N) in a quaternion algebra A, this function returns representatives for the conjugacy classes of maximal orders (or orders of level N) in A.

For definite algebras, the algorithm involves computing the right ideal classes (in fact, the two problems are computationally equivalent in practice).

Example H86E19

In the following example we construct a maximal order in the quaternion algebra ramified at 37, and enumerate its left ideal classes.

```
> S := QuaternionOrder(37);
> ideals := LeftIdealClasses(S);
> [ Basis(I) : I in ideals ];
[
  [ 1, i, j, k ],
  [ 2, 2*i, 1 + i + j, i + k ],
  [ 2, 2*i, i + j, k ]
]
```

We now compute the right orders of the two nontrivial left ideal classes.

```
> _, I, J := Explode(ideals);
> R1 := RightOrder(I);
> R2 := RightOrder(J);
> IsIsomorphic(R1,R2);
true
```

Although the ideals I and J are non-isomorphic left ideals over S, they have isomorphic right orders. In the example following the next section we explore this phenomenon further.

Example H86E20

In this example, we enumerate ideal classes for a definite quaternion algebra (over a totally real field).

```
> P<x> := PolynomialRing(Rationals());
> F<b> := NumberField(x^3-3*x-1);
> Z_F := MaximalOrder(F);
> Foo := InfinitePlaces(F);
> pp := Decomposition(Z_F, 17)[1][1];
> A := QuaternionAlgebra(pp, Foo);
> O := MaximalOrder(A);
> time Rideals := RightIdealClasses(O);
Time: 0.870
> #Rideals;
2
```

Example H86E21

In this example the classgroup of the base field is not trivial and there are some ramified prime ideals in the algebra. Hence we expect nontrival two-sided ideal classes.

```
> K:= QuadraticField(401);
> A:= QuaternionAlgebra< K | -1, -1>;
> RamifiedPlaces(A);
[
    Prime Ideal
    Two element generators:
        2
        $.2 + 1,
    Prime Ideal
    Two element generators:
        2
        $.2 + 2
]
```

```
[ 1st place at infinity, 2nd place at infinity ]
> M:= MaximalOrder(A);
> #TwoSidedIdealClasses(M);
10
> time #RightIdealClasses(M);
140
Time: 6.470
```

86.12.3 Operations on Ideals

In addition to operations on ideals of orders over more general rings (see Section 81.4), the following operations are defined for ideals of quaternion orders over $\mathbf{Z}$, $\mathbf{F}_q[X]$ and number rings.

```
I * J
```

The composite of I and J, where the right order of I equals the left order of J.

```
I meet J
```

Given ideals or orders I and J, this function returns the intersection $I \cap J$.

```
Conjugate(I)
```

Given an ideal I (of a quaternion algebra), this function returns the conjugate ideal.

```
Norm(I)
```

Given an ideal I over $\mathbf{Z}$, this function returns the reduced norm of the ideal, defined as the positive generator of the image of the reduced norm map in $\mathbf{Q}$.

Given an ideal I over $\mathbf{F}_q[X]$, this function returns the reduced norm of the ideal, defined as the normalized generator of the image of the reduced norm map in $\mathbf{F}_q(X)$.

Given an ideal I over a commutative order, this function returns the reduced norm of I as a fractional ideal of the order.

```
Factorization(I)
```

Given a two-sided ideal I of an hereditary order O, this function returns the unique factorization of I into two-sided O-ideals. The result is a sequence of tuples. The first entry of each tuple is a two-sided ideal of O, the second is its exponent. (If the base ring of I is $\mathbf{F}_q[X]$, then q is currently required to be odd.)

86.13 Norm Spaces and Basis Reduction

For definite quaternion orders or ideals one can compute reduced bases and Gram matrices.

If the base ring of the order or ideal is **Z** or $\mathbf{F}_q[X]$ with q odd, the Gram matrices can be made unique up to isomorphism. In fact, in these cases, the isomorphism testing of ideals and orders is based on this reduction.

```
NormSpace(A)
```

Given a quaternion algebra A over any field F not of characteristic 2, this function returns the underlying F-space with inner product the norm form. A map from A into the structure is returned as second value.

```
NormSpace(S)
```

Given a quaternion order S over **Z** or $\mathbf{F}_q[X]$ (with q odd), this function returns the underlying module over its base ring, with inner product respect to the norm. A map from O into the structure is returned as second value.

```
GramMatrix(S)
GramMatrix(I)
```

The Gram matrix of the quaternion order S or ideal I over **Z** or $\mathbf{F}_q[X]$ with respect to the norm on the basis for S.

```
ReducedGramMatrix(S)
```

Given an order or ideal S over **Z** in a definite quaternion algebra, this function returns the Gram matrix G of the corresponding lattice.

The quaternion ideal machinery makes use of a Minkowski basis reduction algorithm which returns a unique normalized reduced Gram matrix G for any definite quaternion ideal. This forms the core of the isomorphism testing for quaternion ideals.

```
ReducedBasis(S)
```

Given an order or ideal S over **Z** in a definite quaternion algebra, this function returns some basis B of S the Gram matrix G of the corresponding lattice associated with S. Note that while there exists a unique Minkowski-reduced Gram matrix G, the basis B is not unique.

Example H86E22

Recall that Minkowski basis reduction is used which returns a unique normalized reduced Gram matrix G for any definite quaternion ideal. This forms the core of the isomorphism testing for quaternion ideals. We illustrate this by applying it to representatives of the set of left ideal classes of an order.

```
> A := QuaternionOrder(19,2);
> ideals := LeftIdealClasses(A);
> #ideals;
5
```

```
> [ (1/Norm(I))*ReducedGramMatrix(I) : I in ideals ];
[
    [ 2  0  1  1]
    [ 0  2  1  1]
    [ 1  1 20  1]
    [ 1  1  1 20],

    [6 0 1 3]
    [0 6 3 1]
    [1 3 8 1]
    [3 1 1 8],

    [6 0 1 3]
    [0 6 3 1]
    [1 3 8 1]
    [3 1 1 8],

    [ 4  0  1 -1]
    [ 0  4  1  1]
    [ 1  1 10  0]
    [-1  1  0 10],

    [ 4  0  1 -1]
    [ 0  4  1  1]
    [ 1  1 10  0]
    [-1  1  0 10]
]
```

`ReducedGramMatrix(S)`

`ReducedBasis(S)`

`Canonical` BoolElt *Default :* `false`

Given an order or ideal S over $\mathbf{F}_q[X]$ in a quaternion algebra A, this function returns a Gram matrix and/or a basis of S whose Gram matrix is in dominant diagonal form (see the function `DominantDiagonalForm` in Section 31.2.1). The Gram matrix will not be unique unless A is definite and `Canonical` is set to `true`.

`ReducedBasis(O)`

`ReducedBasis(I)`

Returns a "reduced" basis for the order O or the ideal I over some number ring. If O or I arise from a definite quaternion algebra, then this basis is LLL-reduced with respect to the norm form; otherwise, the basis is reduced with respect to a Minkowski-like embedding (see [KV10, Section 4]).

```
OptimizedRepresentation(O)
OptimisedRepresentation(O)
```

Given an order O contained in a quaternion algebra A over $\mathbf{Q}$ or a number field F, this function returns a new quaternion algebra A' such that $A' = ((a, b)/F)$ where a and b are small (with respect to O), and, as second return value, an isomorphism $A \to A'$.

```
OptimizedRepresentation(A)
OptimisedRepresentation(A)
```

Given a quaternion algebra A over $\mathbf{Q}$ or a number field F, this function returns a new quaternion algebra A' such that $A' = ((a, b)/F)$ where a and b are small. An isomorphism $A \to A'$ is returned as second value.

```
Enumerate(O, A, B)
```

The sequence of all elements x (up to sign) in the definite quaternion order O or ideal I over $\mathbf{Z}$ such that the reduced norm of x lies in the interval $[A, \ldots B]$ or $[0, \ldots B]$, respectively.

```
Enumerate(O, A, B)
Enumerate(O, B)
```

The sequence of elements x (up to sign) in the definite quaternion order O or ideal I over a number ring such that the absolute trace of the norm of x lies in the interval $[A, \ldots B]$ or $[0, \ldots B]$, respectively.

86.14 Isomorphisms

86.14.1 Isomorphisms of Algebras

Two quaternion algebras A, B over a common field F are isomorphic algebras if and only if they share the same ramified places. Finding an explicit isomorphism is much harder. Currently MAGMA embeds the first standard generator of A into B and then finds another element perpendicular to that image having the correct minimal polynomial. In particular, this requires the construction of two points on a conic (or equivalently, the solution of two norm equations) over quadratic extensions of F.

```
IsIsomorphic(A, B)
```

`Isomorphism` BOOLELT *Default :* `false`

Given two quaternion algebras A, B over $\mathbf{Q}$, $\mathbf{F}_q(X)$ with q odd or a number field, this function returns `true` if and only if they are isomorphic.

If the algebras are isomorphic and `Isomorphism` is set to true, an isomorphism $A \to B$ is also returned.

86.14.2 Isomorphisms of Orders

Two orders S, T in a quaternion algebra A are isomorphic if and only if they are conjugate in A.

In a definite algebra, we use the fact that this conjugation induces an isometry of the quadratic $\mathbf{Z}$- or $\mathbf{F}_q[X]$-modules S and T equipped with the (absolute) norm form. Over an indefinite algebra, we use the fact that any connecting ideal I having left order S and right order T must be isomorphic to a right ideal of T. See [KV10] for details.

```
IsIsomorphic(S, T)
```

```
IsConjugate(S, T)
```

`FindElement`	BoolElt	*Default* : `false`
`ConnectingIdeal`	AlgAssVOrdIdl	*Default* :

Given orders S and T in a quaternion algebra A over $\mathbf{Q}$, $\mathbf{F}_q(X)$ (with q odd) or a number field, this function returns `true` if and only if S and T are isomorphic.

For indefinite algebras, the orders are currently required to be maximal.

If `FindElement` is set, the second return value is an isomorphism from S to T and the third return value is an element a with $T = a^{-1}Sa$, inducing the isomorphism. For indefinite algebras, this search is expensive and may sometimes fail (as in `IsIsomorphic` for ideals, see below).

For indefinite algebras defined over number fields, part of the computation may be faster when a connecting ideal is specified using the parameter `ConnectingIdeal`; this should be an ideal with left order S and right order T.

```
Isomorphism(S, T)
```

Given two isomorphic definite quaternion orders S, T over $\mathbf{Z}$ or $\mathbf{F}_q[X]$ (with q odd), this function returns an algebra isomorphism. For orders over number rings the intrinsic `IsIsomorphic` should be used with the optional argument `FindElement` set.

86.14.3 Isomorphisms of Ideals

Two right (left) ideals I, J of an order O in a quaternion algebra A are isomorphic O-modules if and only if $xI = J$ ($Ix = J$) for some $x \in A^*$.

To decide whether I and J are isomorphic, we test whether the *colon ideal* $(J : I) = \{x \in A : xI \subset J\}$ (similarly defined if I, J are left ideals) is principal, or not.

Over $\mathbf{Z}$, to accomplish this task we compute a Minkowski-reduced Gram matrix of $(J : I)$ which produces an element of smallest norm. Over $\mathbf{F}_q[X]$ we compute a Gram matrix which has dominant diagonal form (see [Ger03]) which also produces a suitable element. Over other number rings R, we search $(J : I)$ for an element of the required norm by computing a reduced basis. This method runs very quickly for "reasonably small" input. (See [KV10, DD08] for further details.)

```
IsIsomorphic(I, J)
```

Given ideals I and J of the same order O in a quaternion algebra A over $\mathbf{Q}$, $\mathbf{F}_q(X)$ (with q odd) or a number field, this function returns `true` if and only if the quaternion ideals I and J are isomorphic (left or right) O-ideals. If I and J are isomorphic, there exists some $x \in A$ such that $xI = J$ (in the case of right O-ideals) or $Ix = J$ (for left ideals). For definite algebras, such an element x is always returned. For indefinite algebras over $\mathbf{Z}$ or a number ring, a search for such an element is made, and if one is found then it is returned.

```
IsPrincipal(I)
```

Given a left (or right) ideal I over $\mathbf{Z}, \mathbf{F}_q[X]$ or a number ring, this function returns `true` if and only if I is a principal ideal; if so, a generator is returned as the second value.

```
IsLeftIsomorphic(I, J)
IsRightIsomorphic(I, J)
```

Given two definite ideals over $\mathbf{Z}$ or $\mathbf{F}_q[X]$ (with q odd) with the same left (or right) order S, this function returns `true` if and only if they are isomorphic as S-modules. The isomorphism and the transforming scalar in the quaternion algebra are returned as second and third values if `true`.

```
IsLeftIsomorphic(I, J)
IsRightIsomorphic(I, J)
```

Given two left (or right) ideals I and J over a number ring, with the same left order O, this function returns `true` if and only if they are isomorphic as O-modules. The isomorphism is given by multiplication as a second return value.

```
LeftIsomorphism(I, J)
```

Given two isomorphic left ideals over a definite order S over $\mathbf{Z}$ or $\mathbf{F}_q[X]$, this function returns the S-module isomorphism between them, followed by the quaternion algebra element which defines the isomorphism by right-multiplication.

```
RightIsomorphism(I, J)
```

Given two isomorphic right ideals over a definite order S over $\mathbf{Z}$ or $\mathbf{F}_q[X]$, this function returns the S-module isomorphism between them, followed by the quaternion algebra element which defines the isomorphism by left-multiplication.

86.14.4 Examples

Example H86E23

In this example, we create two quaternion algebras over $\mathbf{F}_7$, show that they are isomorphic and find an isomorphism between them.

```
> F<x> := RationalFunctionField( GF(7) );
> Q1 := QuaternionAlgebra< F | (x^2+x-1)*(x+1), x >;
> a := x^3 + x^2 + 3;
> b := x^13 + 4*x^11 + 2*x^10 + x^9 + 6*x^8 + 4*x^5 + 3*x^4 + x;
> Q2:= QuaternionAlgebra< F | a, b >;
> ok, phi:= IsIsomorphic(Q1, Q2 : Isomorphism);
> ok;
true
> forall{ <x,y> : x,y in Basis(Q1) |  phi(x*y) eq phi(x)*phi(y) };
true
```

Example H86E24

In this example, we create two ideals, show that they have isomorphic right orders, and then explicitly exhibit the isomorphism.

```
> A := QuaternionAlgebra(37);
> S := MaximalOrder(A);
> ideals := LeftIdealClasses(S);
> _, I, J := Explode(ideals);
> R := RightOrder(I);
> Q := RightOrder(J);
> IsIsomorphic(R,Q);
true
> // Get the x which conjugates R to Q:
> _, pi := Isomorphism(R,Q);
> Norm(pi);
37
> J := lideal< S | [ x*pi : x in Basis(J) ] >;
> RightOrder(J) eq R;
true
```

Example H86E25

We construct two non-isomorphic left ideals with the same left and right orders, then investigate their isomorphisms as right ideals.

```
> S := QuaternionOrder(37);
> ideals := LeftIdealClasses(S);
> _, I, J := Explode(ideals);
> R := RightOrder(I);
> _, pi := Isomorphism(R,RightOrder(J));
> J := lideal< S | [ x*pi : x in Basis(J) ] >;
> IsLeftIsomorphic(I,J);
false
> IsRightIsomorphic(I,J);
true Mapping from: AlgQuatOrd: I to AlgQuatOrd: J given by a rule [no inverse]
1 + i - 2*k
> h, x := RightIsomorphism(I,J);
> y := [1,2,-1,3];
> y := &+[y[i]*b[i] : i in [1 .. 4]] where b is Basis(I);
> h(y);
[-73  15  31   4]
> x*y;
-73 + 15*i + 31*j + 4*k
```

The existence of an isomorphism as a right ideal is due to the fact that the two-sided ideals of R do not have non-isomorphic counterparts in S.

```
> TwoSidedIdealClasses(R);
[ Ideal with basis Pseudo-matrix over Integer Ring
1 * [1 0 0 0]
[0 1 0 0]
[0 0 1 0]
[0 0 0 1]
, Ideal with basis Pseudo-matrix over Integer Ring
1 * [37  0 32 18]
[ 0 37 10  2]
[ 0  0  1  0]
[ 0  0  0  1]
 ]
> TwoSidedIdealClasses(S);
[ Ideal with basis Pseudo-matrix over Integer Ring
1 * [1 0 0 0]
[0 1 0 0]
[0 0 1 0]
[0 0 0 1]
 ]
```

Thus while `Conjugate(I)*J` is in the non-principal R-ideal class, the ideal `I*Conjugate(J)` represents the unique principal ideal class of S.

Example H86E26

We exhibit isomorphism testing for ideals of orders over number rings.

```
> P<x> := PolynomialRing(Rationals());
> F<b> := NumberField(x^3-3*x-1);
> Z_F := MaximalOrder(F);
> F := FieldOfFractions(Z_F);
> A<alpha,beta,alphabeta> := QuaternionAlgebra<F | -3, b>;
> O := Order([alpha,beta,alphabeta]);
> O;
Order of Quaternion Algebra with base ring F
with coefficient ring Maximal Equation Order with defining polynomial x^3 - 3*x
    - 1 over its ground order
> I := ideal<O | 2>;
> I eq (I + ideal<O | 2>);
true
> I eq (I + ideal<O | 3>);
false
>
> Foo := InfinitePlaces(F);
> A := QuaternionAlgebra(ideal<Z_F | 2*3*5>, Foo);
> IsDefinite(A);
true
> O := MaximalOrder(A);
> I := rideal<O | Norm(O.2), O.2>;
> J := rideal<O | Norm(O.3), O.3>;
> IsIsomorphic(I, J);
true (F.2 + F.3) + (27/9190*F.1 - 143/9190*F.2 - 73/9190*F.3)*i +
(-251/27570*F.1 + 7/2757*F.2 + 10/2757*F.3)*k
```

86.15 Units and Unit Groups

Let S be a definite quaternion order over $\mathbf{Z}$, $\mathbf{F}_q[X]$ with q odd, or a number ring. In the first two cases, the unit group of S is finite and can be read off any reduced Gram matrix of S. If the base ring of S is some number ring R, then an explicit description of the finite quotient S^*/R^* is given in [Vig76].

`NormOneGroup(S)`

`ModScalars` BoolElt *Default :* `false`

Returns a group G isomorphic to the group S_1 of elements in S with reduced norm 1 (unless `ModScalars` is set, in which case G is isomorphic to S_1 modulo $\{\pm 1\}$).

The second object returned is a map from G to S expressing the isomorphism.

`Units(S)`

This intrinsic computes the set of units S^* for the definite order S. When the base ring of S is Z or $\mathbf{F}_q[X]$, the returned sequence contains all units of S; when the base field is a number field, the returned sequence contains representatives modulo the unit group of the base ring (since the unit group is infinite in general).

```
MultiplicativeGroup(S)
UnitGroup(S)
```

This intrinsic computes the unit group S^* of the definite quaternion order S. The function returns an abstract group G, and a map from G to S. When the base ring of S is Z or $\mathbf{F}_q[X]$, G represents the full group of units of S; when the base field is a number field, G represents the unit group of S modulo the unit group of the base ring (since the unit group is infinite in general).

Example H86E27

The following example illustrates the unit group computation for an order in a definite quaternion algebra over $\mathbf{Q}$.

```
> A := QuaternionAlgebra< RationalField() | -1, -1 >;
> S1 := MaximalOrder(A);
> S2 := QuaternionOrder(A,2);
> G1, h1 := UnitGroup(S1);
> #G1;
24
> [ A | h1(g) : g in G1 ];
[ 1, -1, -j, -k, i, -1/2 + 1/2*i - 1/2*j - 1/2*k, 1/2 + 1/2*i - 1/2*j + 1/2*k,
-1/2 - 1/2*i - 1/2*j + 1/2*k, -1/2 + 1/2*i + 1/2*j + 1/2*k, 1/2 - 1/2*i - 1/2*j
- 1/2*k, 1/2 + 1/2*i + 1/2*j - 1/2*k, 1/2 - 1/2*i + 1/2*j + 1/2*k, -1/2 - 1/2*i
+ 1/2*j - 1/2*k, 1/2 - 1/2*i + 1/2*j - 1/2*k, -1/2 + 1/2*i + 1/2*j - 1/2*k, 1/2
+ 1/2*i - 1/2*j - 1/2*k, -1/2 - 1/2*i - 1/2*j - 1/2*k, 1/2 + 1/2*i + 1/2*j +
1/2*k, -1/2 - 1/2*i + 1/2*j + 1/2*k, -1/2 + 1/2*i - 1/2*j + 1/2*k, 1/2 - 1/2*i -
1/2*j + 1/2*k, k, -i, j ]
> G2, h2 := UnitGroup(S2);
> #G2;
8
> [ A | h2(g) : g in G2 ];
[ 1, -1, -j, j, k, -k, -i, i ]
```

The unit groups of orders in indefinite quaternion algebras A are infinite arithmetic groups, which are twisted analogues of the groups $\mathrm{SL}_2(\mathbf{Z})$ and their families of subgroups. These are studied in relation to their actions on the upper half complex plane, via an embedding in $\mathrm{GL}_2(\mathbf{R})$ provided by some isomorphism $A \otimes \mathbf{R} \cong M_2(\mathbf{R})$.

Example H86E28

Now we exhibit unit group computations over a number ring.

```
> P<x> := PolynomialRing(Rationals());
> F := NumberField(x^3-3*x-1);
> Z_F := MaximalOrder(F);
> Foo := InfinitePlaces(F);
```

We use SetSeed since the following line makes random choices.

```
> SetSeed(1);
> A := QuaternionAlgebra(ideal<Z_F | 2>, Foo);
> IsDefinite(A);
true
> O := MaximalOrder(A);
> U, h := UnitGroup(O);
> U;
Permutation group U acting on a set of cardinality 12
Order = 12 = 2^2 * 3
    Id(U)
    (1, 2, 4)(3, 6, 7)(5, 9, 10)(8, 12, 11)
    (1, 3)(2, 5)(4, 8)(6, 11)(7, 9)(10, 12)
> #Units(O);
12
```

86.16 Bibliography

[**DD08**] L. Dembele and S. Donnelly. Computing Hilbert Modular Forms Over Fields With Nontrivial Class Group. In S. Pauli F. Hess and M.Pohst, editors, *ANTS VIII*, volume 5011 of *LNCS*. Springer-Verlag, 2008.

[**DG88**] M. Denert and J. Van Geel. The Class Number of Hereditary Orders in Non-Eichler Algebras over Global Fields. *Math. Ann.*, 282:379–393, 1988.

[**Fri97**] Carsten Friedrichs. Berechnung relativer Ganzheitsbasen mit dem Round-2-Algorithmus. Diplomarbeit, Technische Universität Berlin, 1997.
URL:http://www.math.tu-berlin.de/∼kant/publications/diplom/friedrichs.ps.gz.

[**Ger03**] Larry J. Gerstein. Definite quadratic forms over $\mathbf{F}_q[X]$. *J. Algebra*, 268(1):252–263, 2003.

[**IR93**] Gábor Ivanyos and Lajos Rónyai. Finding maximal orders in semisimple algebras over **Q**. *Comput. Complexity*, 3(3):245–261, 1993.

[**KV10**] M. Kirschmer and J. Voight. Algorithmic enumeration of ideal classes for quaternion orders. *SIAM J. Comput. (SICOMP)*, 39(5):1714–1747, 2010.

[**Rei03**] Irving Reiner. *Maximal Orders*, volume 28 of *LMS Monographs.* Oxford University Press, 2003.

[**Vig76**] M.-F. Vignéras. Simplification pour les ordres des corps de quaternions totalement définits. *J. Reine Angew. Math.*, 286-287:257–277, 1976.

[**Vig80**] M.-F. Vignéras. *Arithmétique des Algèbres de Quaternions*, volume 800 of *Lecture Notes in Mathematics.* Springer-Verlag, Berlin, 1980.

[**Voi05**] John Voight. *Quadratic forms and quaternion algebras: Algorithms and arithmetic.* Dissertation, University of California, Berkeley, 2005.

[**Voi11**] John Voight. Identifying the matrix ring: algorithms for quaternion algebras and quadratic forms. 2011.

87 ALGEBRAS WITH INVOLUTION

Chapter 87
ALGEBRAS WITH INVOLUTION

87.1 Introduction

In this chapter we describe techniques for computing with $*$-*algebras*, namely algebras equipped with an anti-automorphism $x \mapsto x^*$ of order at most 2 (an *involution* or *star*). For further information on involutions and the structure of $*$-algebras see [Alb61] and [KMRT98].

The principal application of these techniques is currently to isometry groups of systems of reflexive forms (and the intimately related study of intersections of classical groups). However, it is also possible to use the techniques to compute with group algebras of moderate dimension.

To any set of reflexive forms defined on a common vector space (a *system of forms*) one may associate a matrix $*$-algebra called the *adjoint algebra* of the system. The group of units of this adjoint algebra contains a natural subgroup of *unitary elements*, namely those elements x satisfying the condition $x^* = x^{-1}$. The group of unitary elements coincides with the group of isometries of the system of forms, which is also the intersection of the general classical groups associated with these forms.

The StarAlgebras package provides functions that enable the user to investigate the structure of $*$-algebras. It also provides functions to compute and determine the structure of the group of isometries of a system of reflexive forms, and to compute intersections of arbitrary collections of classical groups defined on a common vector space.

The algorithms are mainly due to Peter Brooksbank and James Wilson [BW11a, BW11b].

87.2 Algebras with Involution

This section introduces two general constructions for $*$-algebras:

(i) The algebra of adjoints of a system of reflexive (alternating, reflexive, or Hermitian) forms $[\phi_1, \ldots, \phi_e]$ defined on a common vector space V.

(ii) The group algebra $K[G]$, where K is any ring and G is a finite group.

We also provide a constructor function for simple $*$-algebras.

87.2.1 Reflexive Forms

A *reflexive form* on a K-vector space V is a bilinear function $\phi : V \times V \to K$ such that, whenever $\phi(u, v) = 0$ for $u, v \in V$, we also have $\phi(v, u) = 0$. Reflexive forms ϕ and ψ on V are *isometric* if $\phi(u, v) = \psi(u, v)$ for all $u, v \in V$; they are *similar* if there exists $a \in K$ such that ϕ and $a\psi$ are isometric. The *radical* of a reflexive form ϕ is the subspace $\mathrm{rad}\phi = \{u \in V : \phi(u, V) = 0\}$, and ϕ is *nondegenerate* if $\mathrm{rad}\phi = 0$.

A fundamental result of Birkhoff and von Neumann states that there are three similarity classes of reflexive forms: *alternating*, *symmetric*, and *Hermitian.* Each such form ϕ is represented by a matrix and an automorphism of K. Specifically, regarding V as the space of row vectors, we specify F and α such that $\phi(u, v) = u^\alpha F v^{\mathrm{tr}}$, where α is the identity if ϕ is bilinear, or the automorphism $x \mapsto \overline{x}$ of order 2 if ϕ is Hermitian. Thus a matrix $g \in \mathrm{GL}(d, K)$ is an *isometry* (respectively *similarity*) of the reflexive form if $g^\alpha F g^{\mathrm{tr}} = F$ (respectively $g^\alpha F g^{\mathrm{tr}} = aF$).

`IsometryGroup(F : parameters)`

`Auto` RngIntElt *Default :* 0

The group of isometries of the (possibly degenerate) reflexive form represented by the matrix F with entries in a finite field $\mathbf{F}_{p^e}$. The field automorphism associated with F is specified by the parameter `Auto`, which represents the exponent f in the map $x \mapsto x^{p^f}$. The default is that F is bilinear on its base ring.

`SimilarityGroup(F : parameters)`

`Auto` RngIntElt *Default :* 0

The group of similarities of the (possibly degenerate) reflexive form represented by the matrix F with entries in a finite field $\mathbf{F}_{p^e}$. The field automorphism associated with F is specified by the parameter `Auto`, which represents the exponent f in the map $x \mapsto x^{p^f}$. The default is that F is bilinear on its base ring.

87.2.2 Systems of Reflexive Forms

A *system of forms* is a sequence $[\phi_1, \ldots, \phi_e]$, where each ϕ_i is a reflexive form on a common K-vector space V. The radical of a system of forms is the intersection of the radicals of the individual forms in the system; A system is *nondegenerate* if its radical is zero.

A classical group on a K-vector space V preserves a reflexive form on V that is unique up to similarity. Hence systems of forms arise naturally from sets of classical groups having V as their common defining module.

Systems of forms also arise naturally from the study of p-groups. Let G be a finite p-group. Let $G = \gamma_1(G) > \gamma_2(G) > \ldots > \gamma_m(G) = 1$ denote the lower central series of G, and let $1 = \zeta_1(G) < \zeta_2(G) < \ldots < \zeta_n(G) = G$ denote its upper central series. Let $\Phi(G)$ be the Frattini subgroup of G, and put $N := \langle \Phi(G), \zeta_{n-1}(G) \rangle$. Then $V = G/N$ and $W = \gamma_2(G)/\gamma_3(G)$ are $\mathrm{GF}(p)$-vector spaces and commutation in G induces a bilinear map $V \times V \to W$. One now obtains a system of forms associated to G by choosing bases for V and W.

Just as matrices are useful representations of bilinear forms, so are systems of forms convenient computational models for *bilinear maps.*

`PGroupToForms(G)`

`PGroupToForms(G)`

Return a system of forms associated to the p-group G. For matrix groups, the input must be a class 2 p-group.

Example H87E1

We construct a system of forms associated to a Sylow 7-subgroup of GL$(3, 7)$.

```
> S := ClassicalSylow(GL (3, 7), 7);
> G := PCGroup(S);
> Forms := PGroupToForms(G);
> Forms;
[
    [0 1]
    [6 0]
]
```

Now we apply the same construction working directly with the matrix group as a p-group of class 2.

```
> Forms := PGroupToForms(S);
> Forms;
[
    [0 1]
    [6 0]
]
```

It is preferable to input a PC-group if such a representation can readily be obtained. The option to use a matrix group for p-groups of class 2 enables the user to construct an associated system of forms in situations where it requires considerably more time to first construct a PC-representation.

87.2.3 Basic Attributes of ∗-Algebras

Using the following functions one can ascertain whether or not a given algebra has an assigned involution, and also access the map if it has.

`IsStarAlgebra(A)`

`IsStarAlgebra(A)`

Return `true` if and only if A has an assigned involution.

`Star(A)`

`Star(A)`

Return the involution associated to the ∗-algebra A.

87.2.4 Adjoint Algebras

Let $S = [\phi_1, \ldots, \phi_e]$ be a system of reflexive forms on a K-vector space V. Let R denote the algebra $\mathrm{End}_K(V)$ and R^{op} denote its opposite ring. Then the *algebra of adjoints* of S is defined as follows:

$$\mathrm{Adj}(S) = \{(x, y) \in R \times R^{\mathrm{op}} : \phi_i(ux, v) = \phi_i(u, yv)\ \forall u, v \in V,\ \forall i \in \{1, \ldots, e\}\}.$$

As the ϕ_i are reflexive forms, $(x, y) \in \mathrm{Adj}(S)$ if and only if $(y, x) \in \mathrm{Adj}(S)$. If S is nondegenerate, then y is uniquely determined by x; thus we identify $\mathrm{Adj}(S)$ with its projection onto R and the assignment $x^* := y$ equips $\mathrm{Adj}(S)$ with an involution.

The function to compute $\mathrm{Adj}(S)$ is an implementation of the algorithms in [BW11a, Proposition 5.1] and [BW11b, Section 5].

`AdjointAlgebra(S : parameters)`

`Autos` SeqEnum *Default :* $[0, .., 0]$

Given a sequence S containing a nondegenerate system of reflexive forms, this function returns the $*$-algebra of adjoints of the forms in S. The parameter `Autos` is used to specify a list of Frobenius exponents associated with the given forms. By default all forms are considered to be bilinear over their common base ring, which must be a finite field. Note that the individual forms in the system are allowed to be degenerate.

Example H87E2

We construct the algebra of adjoints of a particular pair of forms on a vector space of dimension 3 over GF(5^2). The matrices have entries in GF(5) but we regard them as forms over the larger field, and define the first (symmetric) form to be Hermitian over the larger field.

```
> MA := MatrixAlgebra(GF (25), 3);
> F := MA![1,2,0,2,3,4,0,4,1];
> G := MA![0,1,0,4,0,0,0,0,0];
> A := AdjointAlgebra([F, G] : Autos := [1, 0]);
> IsStarAlgebra(A);
true
> Degree(A);
6
> BaseRing(A);
Finite field of size 5
```

Observe that the function has converted the given system of forms to a new system over GF(5) and returned an algebra of degree 6 over the subfield. Now we access the involution on A and apply it to a generator of A.

```
> star := Star(A);
> A.3;
[1 0 2 0 2 0]
[0 1 0 2 0 2]
```

```
[1 0 4 0 4 0]
[0 1 0 4 0 4]
[0 0 0 0 0 0]
[0 0 0 0 0 0]
> A.3@star;
[4 0 3 0 3 0]
[0 4 0 3 0 3]
[4 0 1 0 1 0]
[0 4 0 1 0 1]
[0 0 0 0 0 0]
[0 0 0 0 0 0]
```

87.2.5 Group Algebras

If G is a finite group and R is any ring, then the group algebra $A = R[G]$ possesses a natural involution. Thus each group algebra may be regarded as a $*$-algebra.

`StarOnGroupAlgebra(A)`

The natural involution on the group algebra A induced by inversion on the underlying group. Specifically, if $a = \sum_{g\in G} \alpha_g g \in A$, then $a^* = \sum_{g\in G} \alpha_g g^{-1}$.

`GroupAlgebraAsStarAlgebra(R, G)`

Construct the group algebra $R[G]$ equipped with the natural involution afforded by inversion in G.

Example H87E3

We construct the group algebra $Z[S_3]$ as a $*$-algebra.

```
> G := SymmetricGroup(3);
> K := Integers();
> A := GroupAlgebraAsStarAlgebra(K, G);
> IsStarAlgebra(A);
true;
```

Now access the involution on A and apply it to an element of A.

```
> star := Star(A);
> a := A![0,0,1,0,3,0];
> a;
(1, 3, 2) + 3*(1, 2)
> a@star;
(1, 2, 3) + 3*(1, 2)
```

Alternatively, $Z[S_3]$ can be constructed using the standard constructor and the involution can be attached later.

```
> A := GroupAlgebra(K, G);
> IsStarAlgebra(A);
false
> StarOnGroupAlgebra(A);
Mapping from: AlgGrp: A to AlgGrp: A given by a rule [no inverse]
> IsStarAlgebra(A);
true
```

87.2.6 Simple $*$-Algebras

The Artinian *simple $*$-algebras* – those having no proper $*$-invariant ideals – were classified by Albert [Alb61]. They come in two basic flavours: *classical* and *exchange*. The classical types are simple as algebras and arise as adjoints of nondegenerate reflexive forms. A simple $*$-algebra of exchange type is a direct sum of two isomorphic simple algebras where the involution interchanges the two factors. Naturally extending the Magma classification of reflexive forms (see the manual entry for the function `ClassicalForms`) a simple $*$-algebra S is assigned a `name` as follows:

`"symplectic"` if S is defined by an alternating form;

`"orthogonalcircle"` if S is defined by a symmetric form in odd dimension;

`"orthogonalplus"` if S is defined by a symmetric form of maximal Witt index;

`"orthogonalminus"` if S is defined by a symmetric form of non-maximal Witt index;

`"unitary"` if S is defined by an Hermitian form; and

`"exchange"` if S has exchange type.

Every simple $*$-algebra is isomorphic to a standard simple $*$-algebra having one of these six names defined naturally on a suitable vector space.

We note that involutions on simple $*$-algebras are often classified as being “of the first kind” or “of the second kind” according to whether or not they induce the identity on the center of the algebra. Thus, involutions of the second kind are unitary and exchange, and the others are all involutions of the first kind [KMRT98].

`SimpleStarAlgebra(name, d, K)`

Given a `name`, a positive integer d, and a field K, this function constructs the standard copy of the simple $*$-algebra of type `name` defined naturally on a K-vector space of dimension d.

Example H87E4

We construct the standard copy of the simple $*$-algebra of exchange type on the vector space of dimension 4 over GF(16).

```
> K := GF(16);
> S := SimpleStarAlgebra("exchange", 4, K);
> Dimension(S);
8
> IsStarAlgebra(S);
```

```
true;
> w := K.1;
> s := S.1 * S.2 + w * S.1;
> s;
[   K.1      1      0      0]
[     0      0      0      0]
[     0      0      0      0]
[     0      0      0      0]
> star := Star(S);
> s@star;
[     0      0      0      0]
[     0      0      0      0]
[     0      0    K.1      0]
[     0      0      1      0]
```

87.3 Decompositions of $*$-Algebras

Every finite-dimensional K-algebra A has a decomposition $A = J \oplus W$, where J is the Jacobson radical of A and W is a semisimple subring of A. We refer to such decompositions as *Wedderburn decompositions.* The procedure that computes Wedderburn decompositions is adapted from an analogous MAGMA function written by W. de Graaf for algebras defined by structure constants.

If A is a $*$-algebra and the characteristic of K is not 2, then it follows from a result of Taft [Taf57] that A has a Wedderburn decomposition of the form $A = J \oplus T$ in which T is invariant under the involution of A. We refer to such decompositions as *Taft decompositions.* The procedure that computes Taft decompositions is based on Taft's original proof, and is described in [BW11a, Proposition 4.3].

```
WedderburnDecomposition(A)
```

```
WedderburnDecomposition(A)
```

A Wedderburn decomposition is constructed for the $*$-algebra A. Specifically, the Jacobson radical, J, of A, and a semisimple complement, W, to J in A are computed. Here A may be either a matrix algebra or a group algebra over any field.

```
TaftDecomposition(A)
```

```
TaftDecomposition(A)
```

A Taft decomposition is constructed for the $*$-algebra A. Specifically, the Jacobson radical, J, of A and a $*$-invariant Wedderburn complement to J in A are computed. This function requires that the base ring of A has characteristic different from 2. Here A may be either a matrix $*$-algebra or a group algebra.

Example H87E5

We compute a Wedderburn decomposition of the group algebra GF(5)[A_5] equipped with its natural involution.

```
> K := GF(5);
> G := AlternatingGroup(5);
> A := GroupAlgebraAsStarAlgebra(K, G);
> J, W := WedderburnDecomposition(A);
```

We check dimensions and the $*$-invariance of T.

```
> Dimension(J); Dimension(W);
25
35
> forall { i : i in [1..Ngens (W)] | W.i@Star(A) in W };
false
```

Now find a $*$-invariant decomposition.

```
> J, T := TaftDecomposition(A);
> Dimension(J); Dimension(W);
25
35
> forall { i : i in [1..Ngens(T)] | T.i@Star(A) in T };
true
```

87.4 Recognition of $*$-Algebras

In this section we describe methods that facilitate structural examinations of $*$-algebras. *All of the functions in this section require that the base ring of the given algebra is a finite field of odd order.* The functions are implementations of the methods described in [BW11a, Sections 4.2 and 4.3].

87.4.1 Recognition of Simple $*$-Algebras

If A is a simple $*$-algebra, then we *constructively recognise* A by finding an explicit inverse isomorphisms between A and the standard copy of the simple $*$-algebra which is isomorphic to A. The latter is the output of the function `SimpleStarAlgebra` with the appropriate input parameters.

`RecogniseClassicalSSA(A)`

Given a matrix $*$-algebra A, this function first decides whether or not A is a simple $*$-algebra of classical type. If it is, the standard $*$-algebra, T, corresponding to A, a $*$-isomorphism from A to T, and its inverse from T to A are returned.

`RecogniseExchangeSSA(A)`

Given a matrix $*$-algebra A, this function first decides whether or not A is a simple $*$-algebra of exchange type. If it is, the standard $*$-algebra, T, corresponding to A, a $*$-isomorphism from A to T, and its inverse from T to A are returned.

Example H87E6

We build a particular simple $*$-algebra of symplectic type and recognise it constructively.

```
> MA := MatrixAlgebra(GF(7), 4);
> F := MA![0,1,3,4,6,0,0,1,4,0,0,2,3,6,5,0];
> F;
[0 1 3 4]
[6 0 0 1]
[4 0 0 2]
[3 6 5 0]
> A := AdjointAlgebra([F]);
> isit, T, f, g := RecogniseClassicalSSA(A);
> isit;
true;
```

A quick check that f is, as claimed, a $*$-isomorphism.

```
> (A.1 + A.2)@f eq (A.1@f) + (A.2@f);
true
> (A.1 * A.2)@f eq (A.1@f) * (A.2@f);
true
> (A.2@Star(A))@f eq (A.2@f)@Star(T);
true
```

87.4.2 Recognition of Arbitrary $*$-Algebras

If A is an arbitrary $*$-algebra, then we constructively recognise A as follows:

(i) Find a decomposition $A = J \oplus T$, where J is the Jacobson radical of A and T is a $*$-invariant semisimple complement to J in A;

(ii) Find a decomposition $T = I_1 \oplus \ldots \oplus I_t$ of T into minimal $*$-ideals; and

(iii) For each $j \in \{1, \ldots, t\}$ constructively recognise the simple $*$-algebra I_j.

`RecogniseStarAlgebra(A)`

`RecogniseStarAlgebra(A)`

Constructively recognise the $*$-algebra A given as a matrix $*$-algebra or a group algebra.

There are several functions available that permit easy access to structural information about a $*$-algebra that has been constructively recognised. (In fact all of these functions also initiate a constructive recognition of the input $*$-algebra if the recognition has not already been carried out.) For all of the access functions A can be either a matrix $*$-algebra or a group algebra.

`IsSimpleStarAlgebra(A)`

`IsSimpleStarAlgebra(A)`

Return `true` if and only if A is a simple $*$-algebra.

```
SimpleParameters(A)
```

```
SimpleParameters(A)
```

Given a $*$-algebra A, this function returns the parameters that determine (up to $*$-isomorphism) the minimal $*$-ideals of the semisimple quotient A/J, where J is the Jacobson radical of A. The parameters are returned in the form of a sequence.

```
NormGroup(A)
```

Given a $*$-algebra A, this function returns the group of unitary elements of A, namely the group consisting of all units in A satisfying the condition $x^* = x^{-1}$. The function is based on methods described in [BW11a, Section 5].

Example H87E7

Our first example illustrates how the $*$-algebra machinery may be used to distinguish between group algebras over GF(5) for the dihedral and quaternion groups of order 8. Those group algebras are isomorphic as algebras, but the example shows that they are nonisomorphic as $*$-algebras.

```
> K := GF(5);
> G1 := SmallGroup(8, 3);
> G2 := SmallGroup(8, 4);
> A1 := GroupAlgebraAsStarAlgebra(K, G1);
> A2 := GroupAlgebraAsStarAlgebra(K, G2);
> J1, T1 := TaftDecomposition(A1);
> J2, T2 := TaftDecomposition(A2);
> Dimension(J1); Dimension(J2);
0
0
```

Thus (as we know from Maschke's theorem) both GF(5)[D_8] and GF(5)[Q_8] are semisimple. We now recognise them as $*$-algebras and examine their minimal $*$-ideals.

```
> RecogniseStarAlgebra(A1);
true
> RecogniseStarAlgebra(A2);
true
> SimpleParameters(A1);
[ <"orthogonalcircle", 1, 5>, <"orthogonalcircle", 1, 5>,
<"orthogonalcircle", 1, 5>, <"orthogonalcircle", 1, 5>,
<"orthogonalplus", 2, 5> ]
> SimpleParameters(A2);
[ <"orthogonalcircle", 1, 5>, <"orthogonalcircle", 1, 5>,
<"orthogonalcircle", 1, 5>, <"orthogonalcircle", 1, 5>,
<"symplectic", 2, 5>
]
```

Both group algebras decompose into four 1-dimensional $*$-ideals, and one 4-dimensional $*$-ideal. However, the latter has type `"orthogonalplus"` for GF(5)[D_8], but type `"symplectic"` for GF(5)[Q_8].

Example H87E8

Our second example shows how to use $*$-algebra functions to distinguish between two p-groups of class 2 and order 43^6. The first group is a Sylow 43-subgroup of $\mathrm{GL}(3, 43^2)$.

```
> P1 := ClassicalSylow(GL(3, 43^2), 43);
> Forms1 := PGroupToForms(P1);
> A1 := AdjointAlgebra(Forms1);
> RecogniseStarAlgebra(A1);
true
> SimpleParameters(A1);
[ <"symplectic", 2, 1849> ]
```

The second group is constructed as a subgroup of $\mathrm{GL}(3, \mathrm{GF}(43)[x]/(x^2))$.

```
> R<x> := PolynomialRing(GF(43));
> S, f := quo< R | x^2 >;
> G := GL(3, S);
> Ua := G![1,1,0,0,1,0,0,0,1];
> Wa := G![1,0,0,0,1,1,0,0,1];
> Ub := G![1,x@f,0,0,1,0,0,0,1];
> Wb := G![1,0,0,0,1,x@f,0,0,1];
> P2 := sub< G | [ Ua, Wa, Ub, Wb ] >;
> Forms2 := PGroupToForms(P2);
> A2 := AdjointAlgebra(Forms2);
> RecogniseStarAlgebra(A2);
true
> SimpleParameters(A2);
[ <"symplectic", 2, 43> ]
```

Since A_1 and A_2 are non-isomorphic $*$-algebras, it follows that P_1 and P_2 are non-isomorphic groups.

87.5 Intersections of Classical Groups

The main application of the $*$-algebra machinery is to the study of the group preserving each form in a system of forms; the so-called *isometry group* of the system. An essentially equivalent (but perhaps more familiar) problem is that of computing the intersection of a set of classical groups defined on a common vector space. The main functions are implementations of the algorithms presented in [BW11a, Theorem 1.2] and [BW11b, Theorem 1.1].

`IsometryGroup(S : parameters)`

`Autos` SeqEnum *Default :* $[0, .., 0]$

`DisplayStructure` BoolElt *Default :* `false`

Given a sequence S containing a system of reflexive forms, this function returns the group of isometries of the system. In addition to allowing the individual forms to be degenerate, the function handles degenerate systems.

The field automorphisms associated to the individual forms are specified using the parameter `Autos`; the default is that all forms in the system are bilinear over their common base ring. As well as finding generators for the isometry group, the procedure determines the structure of this group. The parameter `DisplayStructure` may be used to display this structure.

Example H87E9

We compute the isometry group of the system of forms associated to a particular p-group.

```
> G := ClassicalSylow(Sp (4, 5^2), 5);
> S := PGroupToForms(G);
> Parent(S[1]);
Full Matrix Algebra of degree 6 over GF(5)
> I := IsometryGroup(S : DisplayStructure := true);
   G
   |   GL ( 1 , 5 ^ 1 )
   *
   |   5 ^ 4    (unipotent radical)
   1
> #I;
2500
```

`ClassicalIntersection(S)`

Given a sequence S containing a number of classical groups, each one of which preserves (up to similarity) a unique nondegenerate reflexive form on a common finite vector space V, this function returns the intersection of the groups. It is not required that a classical group G in S be the full group of isometries.

Example H87E10

In our final example we intersect two quasisimple classical groups. First we construct a symplectic group $\mathrm{Sp}(F_1)$ for a particular skew-symmetric matrix F_1.

```
> K := GF(3);
> M := UpperTriangularMatrix
>        (K,[0,2,1,0,1,2,1,1,1,2,0,0,1,2,1,0,1,0,1,2,2]);
> F1 := M - Transpose(M);
> G1 := IsometryGroup(F1);
```

First check that G1 is a group of isometries.

```
> forall{ g : g in Generators(G1) | g*F1*Transpose(g) eq F1 };
true
```

Next we construct a quasisimple orthogonal group $\Omega^-(F_2)$ for a particular symmetric matrix F_2.

```
> F2 := SymmetricMatrix
>         (K, [1,2,0,1,2,2,1,0,2,2,1,0,0,0,1,2,1,1,0,1,0]);
> C := TransformForm(F2, "orthogonalminus");
> G := OmegaMinus(6, 3);
> G2 := G^(C^-1);
```

First check that G2 is a group of isometries.

```
> forall { g : g in Generators(G2) | g*F2*Transpose(g) eq F2 };
true
```

Finally compute the intersection of G_1 and G_2 and ask for the order of this intersection group.

```
> I := ClassicalIntersection([G1, G2]);
> #I;
14
```

87.6 Bibliography

[**Alb61**] A. Adrian Albert. *Structure of algebras.* American Mathematical Society, Providence, RI, 1961. Revised printing.

[**BW11a**] Peter A. Brooksbank and James B. Wilson. Computing isometry groups of Hermitian maps. *Transactions of the American Mathematical Society*, 2011. to appear.

[**BW11b**] Peter A. Brooksbank and James B. Wilson. Intersecting two classical groups. Preprint, 2011.

[**KMRT98**] Max-Albert Knus, Alexander Merkurjev, Markus Rost, and Jean-Pierre Tignol. *The book of involutions.* American Mathematical Society, Providence, RI, 1998. Preface by Jacques Tits.

[**Taf57**] E.J. Taft. Invariant Wedderburn factors. *Illinois Journal of Mathematics*, 1:565–573, 1957.

88 CLIFFORD ALGEBRAS

Chapter 88
CLIFFORD ALGEBRAS

88.1 Introduction

This chapter describes Clifford algebras over fields. A Clifford algebra is an associate algebra defined by a quadratic form and so the first part of this chapter describes functions for the creation and manipulation of quadratic spaces; that is, vector spaces with an associated quadratic form. At the moment many of the functions defined for quadratic spaces require the field to be finite and therefore there is not much support for Clifford algebras over infinite fields.

The Clifford algebra of a quadratic form Q defined on a vector space V is an associative algebra C with a vector space homomorphism $f : V \to C$ such that $f(v)^2 = Q(v)$ for all $v \in V$ which satisfies the universal property that if A is any associative algebra with a homomorphism $g : V \to A$ such that $g(v)^2 = Q(v)$ for all $v \in V$, then there is a unique algebra homomorphism $h : C \to A$ such that $hf = g$. It can be shown that f is injective and therefore we may identify V with its image in C. Furthermore, if the dimension of V is n, then the dimension of C is 2^n.

Clifford algebras are represented in Magma as structure constant algebras. The primary reference for quadratic forms and Clifford algebras is [Che97].

88.2 Quadratic Spaces

A quadratic form on a vector space V over a field F is a function $Q : V \to F$ such that

(i) $Q(av) = a^2Q(v)$ for all $a \in F$ and all $v \in V$; and

(ii) the function $\beta : V \times V \to F$ defined by $\beta(u, v) := Q(u+v) - Q(u) - Q(V)$ is a bilinear form on V (called the *polar* form of Q).

A vector space V with an attached quadratic form is called a *quadratic space*. In Magma the polar form of a quadratic space is the Gram matrix B of the space. If the characteristic of the field is not 2, the value of the quadratic form on a row vector v is $\frac{12}{v*B*v^T}$.

A vector $v \neq 0$ is *isotropic* if $\beta(v, v) = 0$; it is defined to be *singular* if $Q(v) = 0$.

The *radical* of V is the subspace of vectors u such that $\beta(u, v) = 0$ for all $v \in V$.

`QuadraticSpace(Q)`

The quadratic space defined by the quadratic form represented by a square matrix Q over a field.

```
QuadraticNorm(v)
```

The value $Q(v)$, where Q is the quadratic form attached to the ambient space of the vector v.

```
IsSingular(V)
```

Determine whether the quadratic space V contains a singular vector; if it does, assign a representative as the second return value.

```
HyperbolicPair(V,u)
```

If u is a singular vector in a quadratic space V this function returns a singular vector v such that $\beta(u, v) = 1$, where β is the polar form of the space, provided u is not in the radical of V.

88.3 Bibliography

[**Che97**] Claude Chevalley. *The algebraic theory of spinors and Clifford algebras*. Springer-Verlag, Berlin, 1997. Collected works. Vol. 2, Edited and with a foreword by Pierre Cartier and Catherine Chevalley, With a postface by J.-P. Bourguignon.

PART XII
REPRESENTATION THEORY

89 MODULES OVER AN ALGEBRA

Chapter 89
MODULES OVER AN ALGEBRA

89.1 Introduction

Let A be an algebra over a field K and let M be a vector space over K. We say that M is a *(right) A-module* if for each $a \in A$ and $m \in M$, a product $ma \in M$ is defined such that

$$(m+n)a = ma + na, m(a+b) = ma + mb,$$
$$m(ab) = (ma)b, m1 = m,$$
$$m(ka) = (ma)k = (mk)a,$$

for all $a, b \in A, m, n \in M, k \in K$.

Recall that a *representation* of an algebra A over a field K is an algebra homomorphism of A into $\mathrm{Hom}_K(M, M)$, for some K-module M. Taking M to be an A-module, and defining a mapping $\rho : M \to M$ by

$$\rho(m) := ma \quad (a \in A, m \in M)$$

then it is an easy exercise to show that ρ is a representation of A. A *matrix representation of degree* n of the algebra A is an algebra homomorphism of A into $M_n(K)$, the complete matrix algebra of degree n over K. Suppose M has finite K-dimension n and choose a basis for M. If, for each $a \in A$, we associate the matrix corresponding to the action of a on the basis elements, we obtain a matrix representation of A. Thus, each A-module of finite K-dimension *affords* a matrix representation of the algebra A. An important special case occurs when $A = K[G]$, the group algebra of a group G. In this case the theory of A-modules coincides with the theory of group representations.

Throughout this chapter we shall use the term *A-module* when referring to modules as defined above. For Magma V2.19, A-modules are restricted to one of the following cases:

(i) A matrix algebra A defined over a field.

(ii) A matrix algebra A defined over a field corresponding to a representation of a group G of finite degree. In this case the user is (implicitly) computing with the group algebra $K[G]$.

(iii) A matrix algebra A defined over an Euclidean Domain R. However, as currently the action of A may be used only in the construction of submodules, discussion will be limited to the case in which the coefficient ring is a field.

Note that in all of the above cases A has finite K-dimension.

Magma provides a range of facilities for defining and computing with A-modules. In particular, extensive machinery is provided for creating $K[G]$-modules which is described in the following chapter. It should be noted however, that many advanced functions only apply when A is an algebra over a finite field. Since the R-module of n-tuples, $R^{(n)}$, underlies an A-module, the operations for $R^{(n)}$ are also applicable.

89.2 Modules over a Matrix Algebra

This section describes function dealing with modules over a matrix algebra, for which there are the most number of operations available.

89.2.1 Construction of an A-Module

89.2.1.1 General Constructions

`RModule(A)`

Given a subalgebra A of $M_n(K)$, create the right A-module M with underlying vector space $K^{(n)}$, where the action of $a \in A$ is given by $m * a$, $m \in M$.

`RModule(Q)`

Given the subalgebra A of $M_n(K)$ generated by the terms of the sequence Q, create the right A-module M with underlying vector space $K^{(n)}$, where the action of $a \in A$ is given by $m * a$, $m \in M$.

Example H89E1

We construct the 6-dimensional module over $\mathbf{F}_2$ with an action given by the matrices

$$\begin{pmatrix} 1 & 0 & 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & 1 & 0 & 1 \\ 0 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 & 0 & 0 \end{pmatrix} \qquad \begin{pmatrix} 1 & 0 & 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 0 & 1 & 1 \\ 0 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix}$$

```
> A := MatrixAlgebra<GF(2), 6 |
>   [ 1,0,0,1,0,1,
>     0,1,0,0,1,1,
>     0,1,1,1,1,0,
>     0,0,0,1,1,0,
>     0,0,0,1,0,1,
>     0,1,0,1,0,0 ],
>   [ 0,1,1,0,1,0,
>     0,0,1,1,1,1,
>     1,0,0,1,0,1,
>     0,0,0,1,0,0,
>     0,0,0,0,1,0,
>     0,0,0,0,0,1 ] >;
> M := RModule(A);
> M;
RModule M of dimension 6 over GF(2)
```

89.2.1.2 Constructions for $K[G]$-Modules

Although $K[G]$-modules are discussed in the next chapter, it is convenient to use them as examples in this chapter and so we give two basic constructions here. The reader is referred to the $K[G]$-module chapter for many other techniques for constructing these modules.

```
GModule(G, Q)
```

`Check` BoolElt *Default :* `true`

Let G be a group defined on r generators and let Q be a sequence of r invertible elements of $M_n(K)$ or $GL(n, K)$. It is assumed that the mapping from G to Q defined by $\phi(G.i) \mapsto Q[i]$, for $i = 1, \ldots, r$, is a group homomorphism from G into $GL(n, K)$. The function constructs a $K[G]$-module M of dimension n, where the action of the generators of G is given by the terms of Q.

```
PermutationModule(G, K)
```

Given a permutation group G and a field K, create the natural permutation module for G over K.

89.2.2 Accessing Module Information

This section deals with the underlying vector space of a module M, which is a module over the algebra A.

89.2.2.1 The Underlying Vector Space

```
M . i
```

Given an A-module M and a positive integer i, return the i-th generator of M.

```
CoefficientRing(M)
BaseRing(M)
```

Given an A-module M, where A is an algebra over the field K, return K.

```
Generators(M)
```

The generators for the A-module M, returned as a set.

```
Parent(u)
```

Given an element u belonging to the A-module M, return M.

89.2.2.2 The Algebra

`Action(M)`

`RightAction(M)`

Given an A-module M, return the matrix algebra A giving the action of A on M.

`MatrixGroup(M)`

`Check` BoolElt *Default :* `true`

Given an $R[G]$-module M, return the matrix group whose generators are the (invertible) generators of the acting algebra of M.

`ActionGenerator(M, i)`

The i-th generator of the (right) acting matrix algebra for the module M.

`NumberOfActionGenerators(M)`

`Ngens(M)`

The number of action generators (the number of generators of the algebra) for the A-module M.

`Group(M)`

Given an $R[G]$-module M, return the group G.

Example H89E2

We illustrate the use of several of these access functions by applying them to the 6-dimensional representation of a matrix algebra defined over $\mathbf{F}_2$.

```
> F2 := GF(2);
> F := MatrixAlgebra(F2, 6);
> A := sub< F |
>   [ 1,0,0,1,0,1,
>     0,1,0,0,1,1,
>     0,1,1,1,1,0,
>     0,0,0,1,1,0,
>     0,0,0,1,0,1,
>     0,1,0,1,0,0 ],
>   [ 0,1,1,0,1,0,
>     0,0,1,1,1,1,
>     1,0,0,1,0,1,
>     0,0,0,1,0,0,
>     0,0,0,0,1,0,
>     0,0,0,0,0,1 ] >;
> T := RModule(F2, 6);
> M := RModule(T, A);
> Dimension(M);
6
```

```
> BaseRing(M);
Finite field of size 2
```

We set R to be the name of the matrix ring associated with M. Using the generator subscript notation, we can access the matrices giving the (right) action of A.

```
> R := RightAction(M);
> R.1;
[1 0 0 1 0 1]
[0 1 0 0 1 1]
[0 1 1 1 1 0]
[0 0 0 1 1 0]
[0 0 0 1 0 1]
[0 1 0 1 0 0]
> R.2;
[0 1 1 0 1 0]
[0 0 1 1 1 1]
[1 0 0 1 0 1]
[0 0 0 1 0 0]
[0 0 0 0 1 0]
[0 0 0 0 0 1]
```

We display full details of the module.

```
> M: Maximal;
Module M of dimension 6 with base ring GF(2)
Generators of acting algebra:

[1 0 0 1 0 1]
[0 1 0 0 1 1]
[0 1 1 1 1 0]
[0 0 0 1 1 0]
[0 0 0 1 0 1]
[0 1 0 1 0 0]

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

89.2.3 Standard Constructions

Given one or more existing modules, various standard constructions are available to construct new modules.

89.2.3.1 Changing the Coefficient Ring

```
ChangeRing(M, S)
```

Given an A-module M with base ring R, together with a ring S, such that there is a natural homomorphism from R to S, construct the module N with base ring S where N is obtained from M by coercing the components of the vectors of M into N. The corresponding homomorphism from M to N is returned as a second value.

```
ChangeRing(M, S, f)
```

Given a module M with base ring R, together with a ring S, and a homomorphism $f : R \to S$, construct the module N with base ring S, where N is obtained from M by applying f to the components of the vectors of M. The corresponding homomorphism from M to N is returned as a second value.

89.2.3.2 Direct Sum

```
DirectSum(M, N)
```

Given R-modules M and N, construct the direct sum D of M and N as an R-module. The embedding maps from M into D and from N into D respectively and the projection maps from D onto M and from D onto N respectively are also returned.

```
DirectSum(Q)
```

Given a sequence Q of R-modules, construct the direct sum D of these modules. The embedding maps from each of the elements of Q into D and the projection maps from D onto each of the elements of Q are also returned.

89.2.3.3 Changing Basis

```
M ^ T
```

Given a $K[G]$-module M of dimension n over the field K, and a nonsingular $n \times n$ matrix T over K, construct the $K[G]$-module N which corresponds to taking the rows of T as a basis for M.

89.2.4 Element Construction and Operations

89.2.4.1 Construction of Module Elements

```
elt< M | a1, ..., an >
```

Given a module M with underlying vector space $K^{(n)}$, and elements $a_1, \ldots, a_n$ belonging to K, construct the element $m = (a_1, \ldots, a_n)$ of M. Note that if m is not an element of M, an error will result.

```
M ! Q
```

Given the module M with underlying vector space K^n, and a sequence $Q = [a_1, \ldots, a_n]$ with universe K, construct the element $m = (a_1, \ldots, a_n)$ of M. Note that if m is not an element of M, an error will result.

```
Zero(M)
```

```
M ! 0
```

The zero element for the A-module M.

```
Random(M)
```

Given a module M defined over a finite ring or field, return a random vector.

89.2.4.2 Deconstruction of Module Elements

```
ElementToSequence(u)
```

```
Eltseq(u)
```

Given an element u belonging to the A-module M, return u in the form of a sequence Q of elements of K.

89.2.4.3 Action of the Algebra on the Module

```
u * a
```

Given a vector u belonging to an A-module M, and an element $a \in A$ return the image of u under the action of a.

```
u * g
```

Given a vector u belonging to an $K[G]$-module M, and an element g belonging to the group G, return the image of u under the action of $K[G]$ on the module M.

89.2.4.4 Arithmetic with Module Elements

```
u + v
```

Sum of the elements u and v, where u and v lie in the same A-module M.

```
-u
```

Additive inverse of the element u.

```
u - v
```

Difference of the elements u and v, where u and v lie in the same A-module M.

```
k * u
```

Given an element u in an A-module M, where A is a K-algebra and an element $k \in K$, return the scalar product $k * u$ as an element of M.

```
u * k
```

Given an element u in an A-module M, where A is a K-algebra and an element $k \in K$, return the scalar product $u * k$ as an element of M.

```
u / k
```

Given an element u in an A-module M, where A is a K-algebra and a non-zero element $k \in K$, return the scalar product $u * (1/k)$ as an element of M.

89.2.4.5 Indexing

```
u[i]
```

Given an element u belonging to a submodule M of the R-module $R^{(n)}$ and a positive integer i, $1 \leq i \leq n$, return the i-th component of u (as an element of the ring R).

```
u[i] := x
```

Given an element u belonging to a submodule M of the R-module $T = R^{(n)}$, a positive integer i, $1 \leq i \leq n$, and an element x of the ring R, redefine the i-th component of u to be x. The parent of u is changed to T (since the modified element u need not lie in M).

89.2.4.6 Properties of Module Elements

```
IsZero(u)
```

Returns `true` if the element u of the A-module M is the zero element.

```
Support(u)
```

A set of integers giving the positions of the non-zero components of the vector u.

89.2.5 Submodules

89.2.5.1 Construction

```
sub< M | L >
```

Given an A-module M, construct the submodule N generated by the elements of M specified by the list L. Each term L_i of the list L must be an expression defining an object of one of the following types:

(a) A sequence of n elements of R defining an element of M;

(b) A set or sequence whose terms are elements of M;

(c) A submodule of M;

(d) A set or sequence whose terms are submodules of M.

The generators stored for N consist of the elements specified by terms L_i together with the stored generators for submodules specified by terms of L_i. Repetitions of an element and occurrences of the zero element are removed (unless N is trivial).

The constructor returns the submodule N as an A-module together with the inclusion homomorphism $f : N \to M$.

`ImageWithBasis(X, M)`

`Check` BoolElt *Default :* `true`

Given a basis matrix X for a A-submodule of the A-module M, return the submodule N of M such that the morphism of N into M is X.

`Morphism(M, N)`

If the A-module M was created as a submodule of the module N, return the inclusion homomorphism $\phi : M \rightarrow N$ as an element of $\mathrm{Hom}_A(M, N)$. Thus, ϕ gives the correspondence between elements of M (represented with respect to the standard basis of M) and elements for N.

Example H89E3

We construct a submodule of the permutation module for $L(3, 4)$ in its representation of degree 21.

```
> G := PSL(3, 4);
> M := PermutationModule(G, GF(2));
> x := M![0,0,0,1,0,1,0,0,0,1,1,0,0,0,1,0,1,1,0,0,1];
> N := sub< M | x >;
> N:Maximal;
GModule N of dimension 9 over GF(2)
Generators of acting algebra:

[1 0 0 0 1 0 1 0 1]
[0 1 0 1 1 1 0 0 0]
[0 0 1 1 1 1 1 0 1]
[0 0 0 0 0 1 1 0 0]
[0 0 0 1 0 0 1 0 0]
[0 0 0 0 1 0 1 0 0]
[0 0 0 1 1 1 0 0 0]
[0 0 0 0 1 1 0 0 1]
[0 0 0 1 0 1 0 1 1]

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

Note that as a $\mathbf{F}_2$-module V has dimension 1, while as a $K[G]$-module it has dimension 9. The submodule N is defined on a reduced basis so we use `Morphism` to see N embedded in M.

```
> phi := Morphism(N, M);
> [ phi(x) : x in Basis(N) ];
[
    M: (1 0 0 0 0 0 0 0 0 1 1 1 1 0 0 0 1 0 1 0 1),
    M: (0 1 0 0 0 0 0 0 0 1 0 0 0 1 0 0 1 1 1 1 1),
    M: (0 0 1 0 0 0 0 0 0 1 0 1 1 0 1 0 1 1 0 1 0),
    M: (0 0 0 1 0 0 0 0 0 0 1 0 1 1 0 1 0 1 1 0 1),
    M: (0 0 0 0 1 0 0 0 0 1 1 0 1 1 1 0 0 0 0 1 1),
    M: (0 0 0 0 0 1 0 0 0 1 0 0 1 1 1 1 1 0 1 0 0),
    M: (0 0 0 0 0 0 1 0 0 0 1 0 0 1 1 1 1 1 0 1 0),
    M: (0 0 0 0 0 0 0 1 0 0 0 1 0 0 1 1 1 1 1 0 1),
    M: (0 0 0 0 0 0 0 0 1 1 1 1 0 0 0 1 0 1 0 1 1)
]
```

89.2.5.2 Membership and Equality

The operators described below refer to the underlying vector space.

`u in M`

Returns `true` if the element u lies in the A-module M.

`N subset M`

Returns `true` if the A-module N is contained in the A-module M.

`N eq M`

Returns `true` if the A-modules N and M are equal, where N and M are contained in a common A-module.

89.2.5.3 Operations on Submodules

`M + N`

Sum of the submodules M and N, where M and N belong to a a common A-module.

`M meet N`

Intersection of the submodules M and N, where M and N belong to a common A-module.

89.2.6 Quotient Modules

`quo< M | L >`

Given an A-module M, construct the quotient module $P = M/N$ as an A-module, where N is the submodule generated by the elements of M specified by the list L. Each term L_i of the list L must be an expression defining an object of one of the following types:

(a) A sequence of n elements of R defining an element of M;

(b) A set or sequence whose terms are elements of M;

(c) A submodule of M;

(d) A set or sequence whose terms are submodules of M.

The generators constructed for N consist of the elements specified by terms L_i together with the stored generators for submodules specified by terms of L_i. The constructor returns the quotient module P as an A-module together with the natural homomorphism $f : M \to P$.

`Morphism(M, N)`

If the A-module N was created as a quotient module of the module M, return the natural homomorphism $\phi : M \to N$ as an element of $\mathrm{Hom}_R(M, N)$. Thus ϕ gives the correspondence between elements of M and elements of N (represented with respect to the standard basis for N).

Example H89E4

We construct a quotient module of the permutation module for $L(3, 4)$ considered above.

```
> G := PSL(3, 4);
> M := PermutationModule(G, GF(2));
> x := M![0,0,0,1,0,1,0,0,0,1,1,0,0,0,1,0,1,1,0,0,1];
> N := sub< M | x >;
> N;
GModule N of dimension 9 over GF(2)
> Q, phi := quo< M | x >;
> Q;
GModule Q of dimension 12 over GF(2)
```

We locate the kernel of the epimorphism ϕ and check that it is the same as N.

```
> K :=  Kernel(phi);
GModule Ker of dimension 9 over GF(2)
> K eq N;
true
```

Given an element x in the codomain Q of the epimorphism ϕ, the value returned as the preimage of x is a representative element of the coset of the kernel that is the actual preimage of x. Since we are working in a module over a finite field, we can explicitly construct the full preimage `PreIm` of x.

```
> x := Q![0,0,0,1,1,0,0,0,0,1,0,0];
> PreIm := { x@@phi + k : k in K };
> #PreIm;
512
```

89.2.7 Structure of a Module

Most of the functions described in this section assume that the base ring is a **finite field**.

89.2.7.1 Reducibility

`Meataxe(M)`

Given an A-module M with base ring a finite field attempt to find a proper submodule N of M or else prove that M is irreducible. If a splitting of M is found, three values are returned:

(a) An A-module N corresponding to the induced action of A on S;

(b) An A-module P corresponding to the induced action of A on the quotient space M/N;

(c) Let ρ, ν and π denote the representations of A afforded by modules M, N and P, respectively. The third value returned is an invertible matrix T which conjugates the matrices of $\rho(A)$ into reduced form. Specifically, if $a \in A$, then

$$T * \rho(a) * T^{-1} = \begin{pmatrix} \nu(a) & 0 \\ * & \pi(a) \end{pmatrix}$$

If M is proved to be irreducible, the function simply returns M. The fact that M is irreducible is recorded as part of the data structure for M.

`IsIrreducible(M)`

Returns `true` if and only if the A-module M is irreducible. If M is reducible, a proper submodule S of M together with the corresponding quotient module $Q = M/S$, are also returned.

`IsAbsolutelyIrreducible(M)`

Returns `true` if and only if the A-module M is absolutely irreducible. Return also a matrix algebra generator for the endomorphism algebra E of M (a field), as well as the dimension of E.

`AbsolutelyIrreducibleModule(M)`

Let A be an algebra over a field K. Given an irreducible A-module M that is not absolutely irreducible over K, return an irreducible module N that is a constituent of the module M considered as a module over the splitting field for M. Note that the module N, while not unique, is absolutely irreducible.

Example H89E5

Consider the group $O_5(3)$ given as a permutation group of degree 45. We construct the permutation module, and we apply the Meataxe manually to find an irreducible constituent.

```
> SetSeed(3);
> O53 := PermutationGroup<45 |
>     (2,3)(4,6)(7,9)(8,11)(12,16)(13,14)(15,19)(18,22)(20,25)(21,26)(27,33)
>       (28,35) (29,34)(31,38)(36,43)(39,41),
>     (1,2,4,7,10,14,16,3,5)(6,8,12,17,21,27,34,41,44)(9,13,18,23,29,37,33,40,43)
>       (11,15,20)(19,24,30,25,31,22,28,36,38)(26,32,39)(35,42,45)>;
>
> P := PermutationModule(O53, GF(2));
> A, B := Meataxe(P); A; B;
GModule A of dimension 20 over GF(2)
GModule B of dimension 25 over GF(2)
> A, B := Meataxe(A); A; B;
GModule A of dimension 14 over GF(2)
GModule B of dimension 6 over GF(2)
> IsIrreducible(A);
true
```

`MinimalField(M)`

Let A be an algebra over a finite field K. Given an A-module M over K, return the smallest subfield of K, over which M can be realised.

`IsPermutationModule(M)`

Returns `true` if and only if the generators of the matrix algebra A are permutation matrices, for a given A-module M.

89.2.7.2 Composition Series

`CompositionSeries(M)`

Given an A-module M, construct a composition series by repeatedly applying the meataxe. The function returns three values:

(a) The composition series as a sequence of A-modules;

(b) The composition factors as a sequence of A-modules in the order determined by the composition series (a);

(c) A transformation matrix t such that for each $a \in A$, $t*a*t^{-1}$ is in reduced form.

`CompositionFactors(M)`

Given an A-module M, construct the composition factors by repeatedly applying the meataxe. The composition factors are returned in the form of a sequence of R-modules in the order determined by a composition series for M. If M is irreducible, the function returns a sequence containing M alone.

```
Constituents(M)
```

Given an A-module M, construct the constituents C of M, i.e., a sequence of representatives for the isomorphism classes of composition factors of M. A sequence I of indices is also returned, so that that i-th element of C is the $I[i]$-th composition factor of M.

```
ConstituentsWithMultiplicities(M)
```

Given an A-module M, return the constituents of M, together with their multiplicities. A sequence I of indices is also returned, so that that i-th element of C is the $I[i]$-th composition factor of M.

Example H89E6

We continue with the $O_5(3)$ example from the previous section. We notice that the constituent of dimension of 8 is not absolutely irreducible, so we lift it to over an extension field.

```
> O53 := PermutationGroup<45 |
>     (2,3)(4,6)(7,9)(8,11)(12,16)(13,14)(15,19)(18,22)(20,25)(21,26)(27,33)
>       (28,35) (29,34)(31,38)(36,43)(39,41),
>     (1,2,4,7,10,14,16,3,5)(6,8,12,17,21,27,34,41,44)(9,13,18,23,29,37,33,40,43)
>       (11,15,20)(19,24,30,25,31,22,28,36,38)(26,32,39)(35,42,45)>;
>
> P := PermutationModule(O53, GaloisField(2));
> Constituents(P);
[
    GModule of dimension 1 over GF(2),
    GModule of dimension 6 over GF(2),
    GModule of dimension 8 over GF(2),
    GModule of dimension 14 over GF(2)
]
> ConstituentsWithMultiplicities(P);
[
    <GModule of dimension 1 over GF(2), 3>,
    <GModule of dimension 6 over GF(2), 1>,
    <GModule of dimension 8 over GF(2), 1>,
    <GModule of dimension 14 over GF(2), 2>
]
> S, F := CompositionSeries(P);
> S, F;
[
    GModule of dimension 14 over GF(2),
    GModule of dimension 20 over GF(2),
    GModule of dimension 21 over GF(2),
    GModule of dimension 29 over GF(2),
    GModule of dimension 30 over GF(2),
    GModule of dimension 31 over GF(2),
    GModule P of dimension 45 over GF(2)
]
```

```
[
    GModule of dimension 14 over GF(2),
    GModule of dimension 6 over GF(2),
    GModule of dimension 1 over GF(2),
    GModule of dimension 8 over GF(2),
    GModule of dimension 1 over GF(2),
    GModule of dimension 1 over GF(2),
    GModule of dimension 14 over GF(2)
]
> IndecomposableSummands(P);
[
    GModule of dimension 1 over GF(2),
    GModule of dimension 44 over GF(2)
]
> C := Constituents(P);
> C;
[
    GModule of dimension 1 over GF(2),
    GModule of dimension 6 over GF(2),
    GModule of dimension 8 over GF(2),
    GModule of dimension 14 over GF(2)
]
> [IsAbsolutelyIrreducible(M): M in C];
[ true, true, false, true ]
> DimensionOfEndomorphismRing(C[3]);
2
> L := GF(2^2);
> E := ChangeRing(C[3], L);
> E;
GModule E of dimension 8 over GF(2^2)
> CE := CompositionFactors(E);
> CE;
[
    GModule of dimension 4 over GF(2^2),
    GModule of dimension 4 over GF(2^2)
]
> IsAbsolutelyIrreducible(CE[1]);
true
> IsIsomorphic(CE[1], CE[2]);
false
```

89.2.7.3 Socle Series

```
IsSemisimple(M)
```

Given an A-module M, which is a $K[G]$-module (`ModGrp`) for a field K, return whether M is semisimple.

If M is a semisimple module defined over a matrix algebra, the function returns as second return value a list of the ranks of the primitive idempotents of the algebra. This is also a list of the multiplicities of the simple modules of the algebra as composition factors in a composition series for the module.

```
MaximalSubmodules(M)
```

Given an A-module M, return a sequence containing the maximal submodules of M.

`Limit` RngIntElt *Default :* 0

If a limit L is provided, only up L submodules are calculated, and the second return value indicates whether all of the submodules are returned.

```
JacobsonRadical(M)
```

The Jacobson radical of the A-module M.

```
MinimalSubmodules(M)
```

Given an A-module M, return a sequence containing the minimal submodules of M.

`Limit` RngIntElt *Default :* 0

If a limit L is provided, only up L submodules are calculated, and the second return value indicates whether all of the submodules are returned.

```
MinimalSubmodules(M, F)
```

Given an A-module M and an irreducible module F, return a sequence containing those minimal submodules of M, each of which is isomorphic to F.

`Limit` RngIntElt *Default :* 0

If a limit L is provided, only up L submodules are calculated, and the second return value indicates whether all of the submodules are returned.

```
MinimalSubmodule(M)
```

Given an A-module M, return a single minimal (or irreducible) submodule of M; if M is itself irreducible, M is returned.

```
Socle(M)
```

Given a A-module M, return its socle, i.e. the sum of the minimal submodules of M.

```
SocleSeries(M)
```

A socle series S for the A-module M, together with the socle factors corresponding to the terms of S and a matrix T giving the transformation of M into (semi-simple) reduced form. The socle series, as returned, does not include the trivial module but does include M.

```
SocleFactors(M)
```

The factors corresponding to the terms of a socle series for the A-module M. The factors are returned in the form of a sequence of A-modules in the order determined by a socle series for M. If M is irreducible, the function returns a sequence containing M alone.

Example H89E7

We continue with the $O_5(3)$ example from the previous section.

```
> O53 := PermutationGroup<45 |
>   (2,3)(4,6)(7,9)(8,11)(12,16)(13,14)(15,19)(18,22)(20,25)(21,26)(27,33)
>     (28,35) (29,34)(31,38)(36,43)(39,41),
>   (1,2,4,7,10,14,16,3,5)(6,8,12,17,21,27,34,41,44)(9,13,18,23,29,37,33,40,43)
>     (11,15,20)(19,24,30,25,31,22,28,36,38)(26,32,39)(35,42,45)>;
>
> P := PermutationModule(O53, FiniteField(2));
> MaximalSubmodules(P);
[
    GModule of dimension 31 over GF(2),
    GModule of dimension 44 over GF(2)
]
> JacobsonRadical(P);
GModule of dimension 30 over GF(2)
> MinimalSubmodules(P);
[
    GModule of dimension 1 over GF(2),
    GModule of dimension 14 over GF(2)
]
> Soc := Socle(P);
> Soc;
GModule Soc of dimension 15 over GF(2)
> SocleSeries(P);
[
    GModule of dimension 15 over GF(2),
    GModule of dimension 22 over GF(2),
    GModule of dimension 30 over GF(2),
    GModule of dimension 31 over GF(2),
    GModule P of dimension 45 over GF(2)
]
> SocleFactors(P);
[
    GModule of dimension 15 over GF(2),
    GModule of dimension 7 over GF(2),
    GModule of dimension 8 over GF(2),
    GModule of dimension 1 over GF(2),
    GModule of dimension 14 over GF(2)
]
```

89.2.8 Decomposability and Complements

The functions in this section currently apply only in the case in which A is an algebra over a finite field.

`IsDecomposable(M)`

Given an A-module M, return `true` iff M is decomposable. If M is decomposable and defined over a finite field, the function also returns proper submodules S and T of M such that $M = S \oplus T$.

`DirectSumDecomposition(M)`

`IndecomposableSummands(M)`

`Decomposition(M)`

Given an A-module M, return a sequence Q of indecomposable summands of M. Each element of Q is an indecomposable submodule of M and M is equal to the (direct) sum of the terms of Q. If M is indecomposable, the sequence Q consists of M alone.

`HasComplement(M, S)`

`IsDirectSummand(M, S)`

Given an A-module M and a submodule S of M, determine whether S has a A-invariant complement in M. If this is the case, the value `true` is returned together with a submodule T of M such that $M = S \oplus T$; otherwise the value `false` is returned.

`Complements(M, S)`

Given an A-module M and a submodule S of M, return all A-invariant complements of S in M.

Example H89E8

```
> A := MatrixAlgebra<GF(2), 6 |
>   [ 1,0,0,1,0,1,
>     0,1,0,0,1,1,
>     0,1,1,1,1,0,
>     0,0,0,1,1,0,
>     0,0,0,1,0,1,
>     0,1,0,1,0,0 ],
>   [ 0,1,1,0,1,0,
>     0,0,1,1,1,1,
>     1,0,0,1,0,1,
>     0,0,0,1,0,0,
>     0,0,0,0,1,0,
>     0,0,0,0,0,1 ] >;
> M := RModule(RSpace(GF(2), 6), A);
> M;
RModule M of dimension 6 over GF(2)
> IsDecomposable(M);
false
> MM := DirectSum(M, M);
> MM;
RModule MM of dimension 12 over GF(2)
> l, S, T := IsDecomposable(MM);
> l;
true;
> S;
RModule S of dimension 6 over GF(2)
> HasComplement(MM, S);
true
> Complements(MM, S);
[
    RModule of dimension 6 over GF(2),
    RModule of dimension 6 over GF(2)
]
> IndecomposableSummands(MM);
[
    RModule of dimension 6 over GF(2),
    RModule of dimension 6 over GF(2)
]
> Q := IndecomposableSummands(MM);
> Q;
[
    RModule of dimension 6 over GF(2),
    RModule of dimension 6 over GF(2)
]
> Q[1] meet Q[2];
RModule of dimension 0 over GF(2)
```

```
> Q[1] + Q[2];
RModule MM of dimension 12 over GF(2)
```

89.2.9 Lattice of Submodules

Let M be an A-module. MAGMA can construct the lattice L of all submodules of M if this is not too large. Various properties of the lattice L may then be examined. The elements of L are called *submodule-lattice elements* and are numbered from 1 to n where n is the cardinality of L. Once the lattice has been constructed, the result of various lattice operations, such as meet and intersection, are available without the need for any module-theoretic calculation. Certain information about M and its submodules may then be obtained by analyzing L. Given an element of L, one can easily create the submodule N of M corresponding to it and one can also create the element of L corresponding to any submodule of M.

The functions is this section currently apply only in the case in which A is an algebra over a finite field.

89.2.9.1 Creating Lattices

`SubmoduleLattice(M)`

`Limit`	RNGINTELT	*Default :* 0
`CodimensionLimit`	RNGINTELT	*Default :* -1

Given an A-module M, construct the lattice L of submodules of M. If a limit n is provided, at most n submodules are calculated, and the second return value indicates whether the returned lattice L is the full lattice of submodules of M.

`SubmoduleLatticeAbort(M, n)`

Given an A-module M and a positive integer n, construct the lattice L of submodules of M, provided that the number of submodule does not exceed n. In this case the value `true` and the lattice L are returned. If M has more than n submodules, the function aborts and returns the value `false`.

`SetVerbose("SubmoduleLattice", i)`

Control verbose printing for the submodule lattice algorithm. The level i can be 2 for maximal printing or 1 for moderate printing. The algorithm works down a composition series of the module and a summary is printed for each level.

`Submodules(M)`

`CodimensionLimit`	RNGINTELT	*Default :* `Dimension(M)`

Given an A-module M, return a sequence containing all submodules of M sorted by dimension.

Example H89E9

We create the lattice of submodules for the A-module $\mathbf{F}_3[\mathbf{Z}_6]$ with level 1 verbose printing turned on.

```
> M := PermutationModule(CyclicGroup(6), GF(3));
> SetVerbose("SubmoduleLattice", 1);
> L := SubmoduleLattice(M);
Submodule Lattice; Dimension: 6, Composition length: 6
Starting level 4; Current number of modules: 2
Starting level 3; Current number of modules: 3
Starting level 2; Current number of modules: 6
Starting level 1; Current number of modules: 9
Starting level 0; Current number of modules: 12
Change basis time: 0.010
Jacobson radical time: 0.060
Complement time: 0.070
Total time: 0.250
> #L;
16
```

89.2.9.2 Operations on Lattices

In the following, L is the lattice of submodules for a module M.

`#L`

The cardinality of L, i.e. the number of submodules of M.

`L ! i`

Create the i-th element of the lattice L. The number i is insignificant (i.e. the elements of L are not numbered in any special way), but this allows one to uniquely identify each element of the lattice L.

`L ! S`

Create the element of the lattice L corresponding to the submodule S of M.

`Bottom(L)`

Create the bottom of the lattice L, i.e. the element of L corresponding to the zero-submodule of M. If the lattice was created with a limit on the number of submodules and the lattice is partial, the bottom of the lattice may not be the zero submodule.

`Random(L)`

Create a random element of L.

`Top(L)`

Create the top of the lattice L, i.e. the element of L corresponding to M.

89.2.9.3 Operations on Lattice Elements

In the following, L is the lattice of submodules for a module M. Elements of L are identified with the integers [1..#L] but not in any particular order.

```
IntegerRing() ! e
```

The integer corresponding to lattice element e.

```
e + f
```

The sum of lattice elements e and f, i.e. the lattice element corresponding to the sum of the modules corresponding to e and f.

```
e meet f
```

The intersection of lattice elements e and f.

```
e eq f
```

Returns `true` if and only if lattice elements e and f are equal.

```
e subset f
```

Returns `true` if and only if e is under f in the lattice L, i.e. the submodule corresponding to e is a submodule of the submodule corresponding to f.

```
MaximalSubmodules(e)
```

The maximal submodules of e, returned as a set of lattice elements.

```
MinimalSupermodules(e)
```

The minimal supermodules of e, returned as a set of lattice elements.

```
Module(e)
```

The submodule of M corresponding to the element e of the lattice L.

89.2.9.4 Properties of Lattice Elements

```
Dimension(e)
```

The dimension of the submodule of M corresponding to e.

```
JacobsonRadical(e)
```

The Jacobson radical of e, i.e. the lattice element corresponding to the Jacobson radical of the submodule corresponding to e.

```
Morphism(e)
```

The morphism from the module corresponding to e to M.

Example H89E10

We create the lattice of submodules for the A-module $\mathbf{F}_3[\mathbf{Z}_6]$.

```
> SetSeed(1);
> M := PermutationModule(CyclicGroup(6), GF(3));
> L := SubmoduleLattice(M);
> #L;
16
> T := Top(L);
> B := Bottom(L);
> T;
16
> B;
1
> // Check that element of L corresponding to M is T
> L ! M;
1
> (L ! M) eq T;
true
> // Check that module corresponding to B is zero-submodule of M
> Module(B);
GModule of dimension 0 with base ring GF(3)
```

We next find the minimal supermodules (immediate parents) of B in L and then determine the actual A-submodules to which they correspond.

```
> S := MinimalSupermodules(B);
> S;
{ 2, 3 }
> Module(L ! 2);
GModule of dimension 1 with base ring GF(3)
> Module(L ! 3);
GModule of dimension 1 with base ring GF(3)
> Dimension(L ! 2);
1
> Morphism(L ! 2);
[1 1 1 1 1 1]
> Morphism(L ! 3);
[1 2 1 2 1 2]
> // Set A to the sum of these elements
> A := L!2 + L!3;
> A;
5;
> // Note that A has dimension 2 and its morphism is the sum of the previous
> Dimension(A);
2
> Morphism(A);
[1 0 1 0 1 0]
[0 1 0 1 0 1]
```

```
> MaximalSubmodules(A);
{ 2, 3}
> S!2 subset A;
true
```

We now find the maximal submodules of L, and examine one, S, in detail.

```
> MaximalSubmodules(T);
{ 14, 15 }
> A := L ! 14;
> Dimension(A);
5
> Morphism(A);
[1 0 0 0 0 1]
[0 1 0 0 0 2]
[0 0 1 0 0 1]
[0 0 0 1 0 2]
[0 0 0 0 1 1]
> S := Module(A);
> S;
GModule S of dimension 5 with base ring GF(3)
```

Finally, we compute the Jacobson radical of S directly, and also obtain it from the lattice, checking that the two methods match.

```
> J := JacobsonRadical(S);
> J;
GModule J of dimension 3 with base ring GF(3)
> L ! J;
8
> JacobsonRadical(A);
8
```

89.2.10 Homomorphisms

Let M and N be A-modules where A is an algebra defined over a field K. Then $\mathrm{Hom}_A(M, N)$ consists of all K-homomorphisms from M to N which commute with the action of A. The type of such (matrix) homomorphisms, called *A-homs*, is `ModMatGrpElt`.

The functions is this section currently apply only in the case in which A is an algebra over a finite field.

89.2.10.1 Creating Homomorphisms

```
hom<  M -> N | X  >
```

Given A-modules M and N, create the (map) homomorphism from M to N given by matrix X.

```
H ! f
```

Given matrix space H, which is $\mathrm{Hom}_A(M,\ N)$ for A-modules M and N, together with a homomorphism f from M to N, create the matrix corresponding to the map f.

```
IsModuleHomomorphism(X)
```

Given a matrix X belonging to $\mathrm{Hom}_K(M, N)$, where M and N are A-modules, return `true` if X is an A-homomorphism.

89.2.10.2 $\mathrm{Hom}(M, N)$

```
Hom(M, N)
```

Given A-modules M and N, construct the vector space of homomorphisms, $\mathrm{Hom}_K(M, N)$, where K is the field over which A is defined.

```
AHom(M, N)
```

Given A-modules M and N, construct the vector space of homomorphisms, $\mathrm{Hom}_A(M, N)$, as a submodule of $\mathrm{Hom}_K(M,\ N)$.

```
GHomOverCentralizingField(M, N)
```

Given A-modules M and N, construct $\mathrm{Hom}_{L[G]}(M,\ N)$ as a subspace of $\mathrm{Hom}_K(M,\ N)$ where L is the centralizing field of M.

Example H89E11

We construct a 12-dimensional module M and a 9-dimensional submodule P of M for a soluble group of order 648 over $\mathbf{F}_3$. We then construct $H = \mathrm{Hom}_A(M, N)$ and perform map operations on elements of H.

```
> G := PermutationGroup< 12 |
>         (1,6,7)(2,5,8,3,4,9)(11,12),
>         (1,3)(4,9,12)(5,8,10,6,7,11) >;
> K := GF(3);
> P := PermutationModule(G, K);
> M := sub< P | [1,0,0,0,0,1,0,0,1,0,0,1] >;
> M;
GModule M of dimension 9 over GF(3)
> H := AHom(P, M);
> H: Maximal;
KMatrixSpace of 12 by 9 GHom matrices and dimension 2 over GF(3)
Echelonized basis:
```

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

[0 0 0 1 1 1 1 1 1]
[0 0 0 1 1 1 1 1 1]
[0 0 0 1 1 1 1 1 1]
[1 1 1 0 0 1 1 1 1]
[1 1 1 0 0 1 1 1 1]
[1 1 1 0 0 1 1 1 1]
[1 1 1 1 1 0 0 1 1]
[1 1 1 1 1 0 0 1 1]
[1 1 1 1 1 0 0 1 1]
[1 1 1 1 1 1 1 0 0]
[1 1 1 1 1 1 1 0 0]
[1 1 1 1 1 1 1 0 0]
> // We write down a random homomorphism from M to P.
> f := 2*H.1 + H.2;
> f;
[2 2 2 1 1 1 1 1 1]
[2 2 2 1 1 1 1 1 1]
[2 2 2 1 1 1 1 1 1]
[1 1 1 2 2 1 1 1 1]
[1 1 1 2 2 1 1 1 1]
[1 1 1 2 2 1 1 1 1]
[1 1 1 1 1 2 2 1 1]
[1 1 1 1 1 2 2 1 1]
[1 1 1 1 1 2 2 1 1]
[1 1 1 1 1 1 1 2 2]
[1 1 1 1 1 1 1 2 2]
[1 1 1 1 1 1 1 2 2]
> Ker := Kernel(f);
> Ker;
GModule Ker of dimension 8 with base ring GF(3)
```

If we print the morphism associated with Ker, we see generators for Ker as a submodule of P.

```
> Morphism(Ker, P);
[1 0 2 0 0 0 0 0 0 0 0 0]
[0 1 2 0 0 0 0 0 0 0 0 0]
```

```
[0 0 0 1 0 2 0 0 0 0 0 0]
[0 0 0 0 1 2 0 0 0 0 0 0]
[0 0 0 0 0 0 1 0 2 0 0 0]
[0 0 0 0 0 0 0 1 2 0 0 0]
[0 0 0 0 0 0 0 0 0 1 0 2]
[0 0 0 0 0 0 0 0 0 0 1 2]
> // Examine the image of f and its morphism to P.
> Im := Image(f);
> Im;
GModule Im of dimension 4 with base ring GF(3)
> Morphism(Im, P);
[1 1 1 0 0 0 0 0 0 0 0 0]
[0 0 0 1 1 1 0 0 0 0 0 0]
[0 0 0 0 0 0 1 1 1 0 0 0]
[0 0 0 0 0 0 0 0 0 1 1 1]
```

Example H89E12

We construct a G-homomorphism module H_1 for a G-module and then the homomorphism module $H = \mathrm{Hom}(H_1, H_1)$ with right matrix action which is equivalent to (the right representation of) the endomorphism module of H.

```
> P := GModule(CyclicGroup(11), GF(3));
> F := Constituents(P);
> F;
[
    GModule of dimension 1 over GF(3),
    GModule of dimension 5 over GF(3),
    GModule of dimension 5 over GF(3)
]
> H1 := GHom(P, F[2]);
> H1;
KMatrixSpace of 2 by 3 matrices and dimension 1 over Rational Field
> H := Hom(H1, H1, "right");
> H: Maximal;
KMatrixSpace of 5 by 5 matrices and dimension 5 over GF(3)
Echelonized basis:

[1 0 0 0 0]
[0 1 0 0 0]
[0 0 1 0 0]
[0 0 0 1 0]
[0 0 0 0 1]

[0 1 0 0 0]
[1 1 1 2 1]
[2 0 2 1 1]
[2 1 0 0 0]
```

```
[0 2 1 0 0]

[0 0 1 0 0]
[2 0 2 1 1]
[2 2 2 2 2]
[2 0 1 0 2]
[1 0 1 2 1]

[0 0 0 1 0]
[2 1 0 0 0]
[2 0 1 0 2]
[2 2 1 2 2]
[2 1 1 0 1]

[0 0 0 0 1]
[0 2 1 0 0]
[1 0 1 2 1]
[2 1 1 0 1]
[2 1 0 0 2]
```

89.2.10.3 Endo– and Automorphisms

`EndomorphismAlgebra(M)`

`EndomorphismRing(M)`

`Direct` BoolElt *Default :* `false`

Given a A-module M with base ring K, construct $E = \mathrm{End}_A(M)$ as a subring E of the complete matrix ring $K^{(n\times n)}$.

`CentreOfEndomorphismRing(M)`

Given a A-module M with base ring K, construct the centre of $\mathrm{End}_A(M)$ as a subring Z of the complete matrix ring $K^{(n\times n)}$. This is equivalent to `Centre(EndomorphismRing(M))` but will often be much faster.

`AutomorphismGroup(M)`

Given a A-module M with base ring K, construct $\mathrm{Aut}(M)$ as a subgroup G of the general linear group $GL(n, K)$. Thus, G is the group of units of $\mathrm{End}(M)$.

`IsIsomorphic(M, N)`

Returns `true` if the A-modules M and N are isomorphic, `false` otherwise. If M and N are isomorphic, the function also returns a matrix T such that $M^T = N$. Note that the action generators of M and N must match, so the function effectively determines whether there is an invertible matrix T such that `T^-1*ActionGenerator(M, i)*T` equals `ActionGenerator(N, i)` for each i.

Example H89E13

We construct the endomorphism ring for a permutation module over $\mathbf{F}_3$ for a soluble group of order 648.

```
> G := PermutationGroup< 12 |
>         (1,6,7)(2,5,8,3,4,9)(11,12),
>         (1,3)(4,9,12)(5,8,10,6,7,11) >;
> P := PermutationModule(G, GF(3));
> time End := EndomorphismAlgebra(P);
Time: 0.000
> End;
Matrix Algebra of degree 12 and dimension 3 over GF(3)
```

Thus, the permutation module P has 27 endomorphisms.

```
> time Aut := AutomorphismGroup(P);
Time: 0.010
> Aut;
MatrixGroup(12, GF(3))
Generators:
    [1 0 0 1 1 1 1 1 1 1 1 1]
    [0 1 0 1 1 1 1 1 1 1 1 1]
    [0 0 1 1 1 1 1 1 1 1 1 1]
    [1 1 1 1 0 0 1 1 1 1 1 1]
    [1 1 1 0 1 0 1 1 1 1 1 1]
    [1 1 1 0 0 1 1 1 1 1 1 1]
    [1 1 1 1 1 1 1 0 0 1 1 1]
    [1 1 1 1 1 1 0 1 0 1 1 1]
    [1 1 1 1 1 1 0 0 1 1 1 1]
    [1 1 1 1 1 1 1 1 1 1 0 0]
    [1 1 1 1 1 1 1 1 1 0 1 0]
    [1 1 1 1 1 1 1 1 1 0 0 1]

    [2 1 1 0 0 0 0 0 0 0 0 0]
    [1 2 1 0 0 0 0 0 0 0 0 0]
    [1 1 2 0 0 0 0 0 0 0 0 0]
    [0 0 0 2 1 1 0 0 0 0 0 0]
    [0 0 0 1 2 1 0 0 0 0 0 0]
    [0 0 0 1 1 2 0 0 0 0 0 0]
    [0 0 0 0 0 0 2 1 1 0 0 0]
    [0 0 0 0 0 0 1 2 1 0 0 0]
    [0 0 0 0 0 0 1 1 2 0 0 0]
    [0 0 0 0 0 0 0 0 0 2 1 1]
    [0 0 0 0 0 0 0 0 0 1 2 1]
    [0 0 0 0 0 0 0 0 0 1 1 2]

    [0 1 1 0 0 0 0 0 0 0 0 0]
    [1 0 1 0 0 0 0 0 0 0 0 0]
    [1 1 0 0 0 0 0 0 0 0 0 0]
    [0 0 0 0 1 1 0 0 0 0 0 0]
    [0 0 0 1 0 1 0 0 0 0 0 0]
    [0 0 0 1 1 0 0 0 0 0 0 0]
    [0 0 0 0 0 0 0 1 1 0 0 0]
    [0 0 0 0 0 0 1 0 1 0 0 0]
    [0 0 0 0 0 0 1 1 0 0 0 0]
    [0 0 0 0 0 0 0 0 0 0 1 1]
    [0 0 0 0 0 0 0 0 0 1 0 1]
    [0 0 0 0 0 0 0 0 0 1 1 0]
> #Aut;
18
> IsAbelian(Aut);
true
> AbelianInvariants(Aut);
[ 3, 6 ]
```

The module has 18 automorphisms. The automorphism group is isomorphic to the abelian group $\mathbf{Z}_2 \times \mathbf{Z}_3 \times \mathbf{Z}_3$.

89.3 Modules over a General Algebra

89.3.1 Introduction

This section describes the functionality for modules over general algebras in Magma. A left-module over an algebra A is a module M together with a bilinear map $A \times M \to M$. A right-module over A is a module M together with a bilinear map $M \times A \to M$. Magma provides functionality for both kinds of modules.

89.3.2 Construction of Algebra Modules

`Module(A, m)`

For an algebra A this function creates a module over A. If the module will be a left-module then m is a map from the Cartesian product $A \times M$ to M. If the module will be a right-module then m is a map from $M \times A$ to M. Here M has to be an R-module, where R is the coefficient field of A.

Example H89E14

We create the right-module over the full matrix algebra of 3×3 - matrices acting on its natural module.

```
> A:= MatrixAlgebra(Rationals(), 3);
> V:= RModule(Rationals(), 3);
> m:= map< CartesianProduct(V, A) -> V | t :-> t[1]*t[2] >;
> Module(A, m);
Right Module of Full Matrix Algebra of degree 3 over Rational Field
```

89.3.3 The Action of an Algebra Element

```
a ^ v
```

Given an element v of a left-module over an algebra A, and an element a of A computes the result of letting a act on v.

```
v ^ a
```

Given an element v of a right-module over an algebra A and an element a of A computes the result of letting a act on v.

```
ActionMatrix(M, a)
```

Given a module M over an algebra A and an element a of A returns the matrix of the action of a on M. If M is a left-module then the i-th column of this matrix contains the coordinates of the image of a acting on the i-th basis element of M. If A is a right-module then the rows contain these coordinates.

Example H89E15

```
> A:= MatrixAlgebra(Rationals(), 3);
> V:= RModule(Rationals(), 3);
> m:= map< CartesianProduct(V, A) -> V | t :-> t[1]*t[2] >;
> M:=Module(A, m);
> M.1^A.1;
M: (1 0 0)
> ActionMatrix(M, A.2);
[0 1 0]
[0 0 1]
[1 0 0]
```

89.3.4 Related Structures of an Algebra Module

```
Algebra(M)
```

This returns the algebra over which the algebra module M is defined.

```
CoefficientRing(M)
```

Returns the ground field of the algebra module M.

```
Basis(M)
```

Returns a sequence containing the basis vectors of the algebra module M.

89.3.5 Properties of an Algebra Module

```
IsLeftModule(M)
```

This returns `true` if the algebra module M is a left-module, and `false` if it is a right module.

```
IsRightModule(M)
```

This returns `true` if the algebra module M is a right-module, and `false` if it is a left module.

```
Dimension(M)
```

The dimension of the algebra module M.

89.3.6 Creation of Algebra Modules from other Algebra Modules

```
DirectSum(Q)
```

Given a sequence Q of algebra modules (all defined over the same algebra, and all left (respectively right) modules), returns the module M that is the direct sum of the modules in Q. Furthermore, two sequences of mappings are returned. The i-th element of the first sequence is the embedding of the i-th element of Q into M. The i-th element of the second sequence is the projection of M onto the i-th element of Q.

```
SubalgebraModule(B, M)
```

Given an algebra module M over the algebra A, and a subalgebra B of A, return M as a B-module.

```
ModuleWithBasis(Q)
```

Given a sequence Q containing the elements of a particular basis of an algebra module M, create an algebra module that is isomorphic to M, but with basis Q. (Or, more precisely, the basis vectors of the module V that is returned are in bijection with Q. The action of an algebra element on the i-th basis vector of V is computed by computing it on the i-th vector in Q and expressing the result as a linear combination of the elements of Q. The resulting coordinates are used to form the corresponding element of V.) This can be used to compute the action of algebra elements with respect to a given basis of M.

Example H89E16

```
> A:= MatrixAlgebra(Rationals(), 3);
> V:= RModule(Rationals(), 3);
> m:= map< CartesianProduct(V, A) -> V | t :-> t[1]*t[2] >;
> M:=Module(A, m);
> N:=DirectSum([ M, M ]);
> ActionMatrix(N, A.1);
[1 0 0 0 0 0]
[0 0 0 0 0 0]
[0 0 0 0 0 0]
[0 0 0 1 0 0]
[0 0 0 0 0 0]
[0 0 0 0 0 0]
> W:= ModuleWithBasis([ M.1+M.2+M.3, M.2+M.3, M.3 ]);
> ActionMatrix(W, A.1);
[ 1 -1  0]
[ 0  0  0]
[ 0  0  0]
```

```
sub< M | S >
```

```
sub< M | e1, ..., en >
```

Return the submodule of M containing the elements in the sequence S or the elements $e1$, …, en.

```
quo< M | S >
```

```
quo< M | e1, ..., en >
```

Construct the quotient module of M by the submodule S of M, the submodule containing the elements in the sequence S or the elements $e1, ..., en$.

90 $K[G]$-MODULES AND GROUP REPRESENTATIONS

Chapter 90

$K[G]$-MODULES AND GROUP REPRESENTATIONS

90.1 Introduction

A module over a group algebra, $K[G]$, where K is a field and G is a group, is an important special case of modules over an algebra. This case coincides with the theory of group representations. MAGMA provides extensive machinery for constructing $K[G]$-modules. It should be noted however, that some advanced functions apply only when K is a finite field.

In this chapter the machinery for constructions peculiar to $K[G]$-modules will be described. In addition, a number of operations that apply only to $K[G]$-modules are described. All of the operations for A-modules also apply to $K[G]$-modules and are not repeated in this chapter.

90.2 Construction of $K[G]$-Modules

The following functions provide for the construction of finite-dimensional $K[G]$-modules for a group G, where the action of G is given in terms of a matrix representation of G. Note that an Euclidean Domain may appear in place of the field K.

90.2.1 General $K[G]$-Modules

```
GModule(G, A)
```

Let G be a group defined on r generators and let A be a subalgebra of the matrix algebra $M_n(K)$, also defined by r non-singular matrices. It is assumed that the mapping from G to A defined by $\phi(G.i) \mapsto A.i$, for $i = 1, \ldots, r$, is a group homomorphism. Let M be an n-dimensional vector space over K. The function constructs a $K[G]$-module M of dimension n, where the action of the i-th generator of G on M is given by the i-th generator of A.

```
GModule(G, Q)
```

Let G be a group defined on r generators and let Q be a sequence of r invertible elements of $M_n(K)$ or $GL(n, K)$. It is assumed that the mapping from G to Q defined by $\phi(G.i) \mapsto Q[i]$, for $i = 1, \ldots, r$, is a group homomorphism from G into the matrix algebra A defined by the terms of Q. The function constructs a $K[G]$-module M of dimension n, where the action of G is defined by the matrix algebra A.

```
TrivialModule(G, K)
```

Create the trivial $K[G]$-module for the group G.

Example H90E1

We construct a 3-dimensional module for PSL(2, 7) over $\mathbf{F}_2$. The action of the group on M is described in terms of two elements, x and y, belonging to the ring of 3×3 matrices over $\mathbf{F}_2$.

```
> PSL27 := PermutationGroup< 8 | (2,3,5)(6,7,8), (1,2,4)(3,5,6) >;
> S := MatrixAlgebra< FiniteField(2), 3 |
>        [ 0,1,0,  1,1,1,  0,0,1 ], [ 1,1,1,  0,1,1,  0,1,0 ] >;
> M := GModule(PSL27, S);
> M: Maximal;
GModule M of dimension 3 with base ring GF(2)
Generators of acting algebra:

[0 1 0]
[1 1 1]
[0 0 1]

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

Example H90E2

We write a function which, given a matrix algebra A, together with a matrix group G acting on A by conjugation (so A is closed under action by G), computes the G-module M representing the action of G on A. We then construct a particular (nilpotent) upper-triangular matrix algebra A, a group $G = 1 + A$ which acts on A, and finally construct the appropriate G-module M.

```
> MakeMod := function(A, G)
>     // Make G-module M of G acting on A by conjugation
>     k := CoefficientRing(A);
>     d := Dimension(A);
>     S := RMatrixSpace(A, k);
>     return GModule(
>         G,
>         [
>             MatrixAlgebra(k, d) |
>             &cat[
>                 Coordinates(S, S.j^g): j in [1 .. d]
>             ] where g is G.i: i in [1 .. Ngens(G)]
>         ]
>     );
> end function;
>
> MakeGroup := function(A)
>     // Make group G from upper-triangular matrix algebra A
>     k := CoefficientRing(A);
>     n := Degree(A);
>     return MatrixGroup<n, k | [Eltseq(1 + A.i): i in [1 .. Ngens(A)]]>;
> end function;
>
> k := GF(3);
> n := 4;
> M := MatrixAlgebra(k, n);
> A := sub<M |
>      [0,2,1,1, 0,0,1,1, 0,0,0,1, 0,0,0,0],
>      [0,1,0,0, 0,0,2,2, 0,0,0,1, 0,0,0,0]>;
> G := MakeGroup(A);
> G;
MatrixGroup(4, GF(3)) of order 3^4
Generators:
    [1 2 1 1]
    [0 1 1 1]
    [0 0 1 1]
    [0 0 0 1]

    [1 1 0 0]
    [0 1 2 2]
    [0 0 1 1]
    [0 0 0 1]
> M := MakeMod(A, G);
> M: Maximal;
GModule M of dimension 5 over GF(3)
Generators of acting algebra:

[1 1 1 2 0]
[0 1 1 0 0]
[0 0 1 0 0]
[0 0 0 1 0]
[0 0 0 0 1]

[1 2 2 2 0]
[0 1 1 0 0]
[0 0 1 0 0]
[0 0 0 1 0]
[0 0 0 0 1]
```

90.2.2 Natural $K[G]$-Modules

The following functions provide for the construction of $K[G]$-modules for a group G in one of its natural actions. Note that an Euclidean Domain may be used in place of the field K.

```
GModule(G, K)
```

Given a finite permutation group G and a ring K, create the natural permutation module for G over K.

```
GModule(G)
```

Given a matrix group G defined as a subgroup of the group of units of the ring $Mat_n(K)$, where K is a field, create the natural $K[G]$-module for G.

Example H90E3

Given the Mathieu group M_{11} presented as a group of 5×5 matrices over $\mathbf{F}_3$, we construct the natural $K[G]$-module associated with this representation.

```
> G := MatrixGroup<5, FiniteField(3) |
>      [ 2,1,2,1,2,  2,0,0,0,2,  0,2,0,0,0,  0,1,2,0,1,  1,0,2,2,1],
>      [ 2,1,0,2,1,  1,2,0,2,2,  1,1,2,1,1,  0,2,0,1,1,  1,1,2,2,2] >;
> Order(G);
7920
>
> M := GModule(G);
> M : Maximal;
GModule M of dimension 5 with base ring GF(3)
Generators of acting algebra:

[2 1 2 1 2]
[2 0 0 0 2]
[0 2 0 0 0]
[0 1 2 0 1]
[1 0 2 2 1]

[2 1 0 2 1]
[1 2 0 2 2]
[1 1 2 1 1]
[0 2 0 1 1]
[1 1 2 2 2]
```

90.2.3 Action on an Elementary Abelian Section

```
GModule(G, A, B)
```

```
GModule(G, A)
```

Given a group G, a normal subgroup A of G and a normal subgroup B of A such that the section A/B is elementary abelian of order p^n, create the $K[G]$-module M corresponding to the action of G on A/B, where K is the field $\mathbf{F}_p$. If B is trivial, it may be omitted. The function returns

(a) the module M; and

(b) the homomorphism $\phi : A/B \to M$.

Example H90E4

We construct a module M for the wreath product G of the alternating group of degree 4 with the cyclic group of degree 3. The module is given by the action of G on an elementary abelian normal subgroup H of order 64.

```
> G := WreathProduct(AlternatingGroup(4), CyclicGroup(3));
> G := PCGroup(G);
> A := pCore(G, 2);
> A;
GrpPC of order 64 = 2^6
Relations:
A.1^2 = Id(A),
A.2^2 = Id(A),
A.3^2 = Id(A),
A.4^2 = Id(A),
A.5^2 = Id(A),
A.6^2 = Id(A)
> M := GModule(G, A, sub<G|>);
> M;
GModule of dimension 6 with base ring GF(2)
```

90.2.4 Permutation Modules

The following functions provide for the construction of permutation modules for a group G. Note that an Euclidean Domain may be used in place of the field K.

```
PermutationModule(G, H, K)
```

Given a group G, a subgroup H of finite index in G and a field K, create the $K[G]$-module for G corresponding to the permutation action of G on the cosets of H.

```
PermutationModule(G, K)
```

Given a permutation group G and a field K, create the natural permutation module for G over K.

```
PermutationModule(G, V)
```

Given a permutation group G of degree n and an n-dimensional vector space V, create the natural permutation module for G over K.

```
PermutationModule(G, u)
```

Given a permutation group G of degree n, and a vector u belonging to the vector space $V = K^{(n)}$, construct the $K[G]$-module corresponding to the action of G on the K-subspace of V generated by the set of vectors obtained by applying the permutations of G to the vector u.

Example H90E5

We construct the permutation module for the Mathieu group M_{12} over the field $\mathbf{F}_2$.

```
> M12 := PermutationGroup<12 |
>       (1,2,3,4,5,6,7,8,9,10,11),
>       (1,12,5,2,9,4,3,7)(6,10,11,8) >;
> M := PermutationModule(M12, FiniteField(2));
> M : Maximal;
GModule M of dimension 12 with base ring GF(2)
Generators of acting algebra:

[0 1 0 0 0 0 0 0 0 0 0 0]
[0 0 1 0 0 0 0 0 0 0 0 0]
[0 0 0 1 0 0 0 0 0 0 0 0]
[0 0 0 0 1 0 0 0 0 0 0 0]
[0 0 0 0 0 1 0 0 0 0 0 0]
[0 0 0 0 0 0 1 0 0 0 0 0]
[0 0 0 0 0 0 0 1 0 0 0 0]
[0 0 0 0 0 0 0 0 1 0 0 0]
[0 0 0 0 0 0 0 0 0 1 0 0]
[0 0 0 0 0 0 0 0 0 0 1 0]
[1 0 0 0 0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0 0 0 0 1]

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

Example H90E6

We construct the constituent of the permutation module for the alternating group of degree 7 that contains the vector $(1, 0, 1, 0, 1, 0, 1)$.

```
> A7 := AlternatingGroup(7);
> V  := VectorSpace(FiniteField(2), 7);
> x  := V![1,0,1,0,1,0,1];
> M  := PermutationModule(A7, x);
> M : Maximal;
GModule of dimension 6 with base ring GF(2)
```

```
Generators of acting algebra:

[1 0 1 0 0 0]
[0 0 1 0 1 0]
[0 1 1 0 0 0]
[0 0 1 0 0 0]
[0 0 1 1 0 0]
[0 0 1 0 0 1]

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

90.2.5 Action on a Polynomial Ring

```
GModule(G, P, d)
```

Let G be a permutation group of degree n or a or matrix group of degree n over a finite field, $P = K[x_1, \ldots, x_n]$ a polynomial ring over a field K in n variables, and d a non-negative integer. This function creates the $K[G]$-module M corresponding to the action of G on the space of homogeneous polynomials of degree d of the polynomial ring P. The function also returns the isomorphism f between the space of homogeneous polynomials of degree d of P and M, together with an indexed set of monomials of degree d of P which correspond to the columns of M.

```
GModule(G, I, J)
```

Let G be a permutation group of degree n or a or matrix group of degree n over a finite field, I an ideal of a multivariate polynomial ring $P = K[x_1, \ldots, x_n]$ in n variables over a field K, and J a zero-dimensional subideal of I. This function creates the $K[G]$-module M corresponding to the action of G on the finite-dimensional quotient I/J. The function also returns the isomorphism f between the quotient space I/J and M, together with an indexed set of monomials of P, forming a (vector space) basis of I/J, and which correspond to the columns of M.

```
GModule(G, Q)
```

Let G be a permutation group of degree n or a or matrix group of degree n over a finite field and $Q = I/J$ a finite-dimensional quotient ring of a multivariate polynomial ring $P = K[x_1, \ldots, x_n]$ in n variables over a field K. This function creates the $K[G]$-module M corresponding to the action of G on the finite-dimensional quotient Q. The function also returns the isomorphism f between the quotient ring Q and M, together with an indexed set of monomials of P, forming a (vector space) basis of Q, and which correspond to the columns of M.

Example H90E7

Let T be the polynomial ring in five indeterminates over $GF(5)$. We create the representation of the alternating group of degree 5 that corresponds to its action on the space H_4 of homogeneous polynomials of degree 4 of T.

```
> G := Alt(5);
> R<[x]> := PolynomialRing(GF(5), 5);
> M, f := GModule(G, R, 4);
> M;
GModule M of dimension 70 over GF(5)
```

Thus, the action of $Alt(5)$ on H_4 yields a 70-dimensional module. We find its irreducible constituents.

```
> Constituents(M);
[
    GModule of dimension 1 over GF(5),
    GModule of dimension 3 over GF(5),
    GModule of dimension 5 over GF(5)
]
> t := x[1]^4 + x[2]^4 + x[3]^4 + x[4]^4 + x[5]^4;
> v := f(t); v;
M: (1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
    1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 1 0 0 0 1)
> v@@f;
x[1]^4 + x[2]^4 + x[3]^4 + x[4]^4 + x[5]^4
```

90.3 The Representation Afforded by a $K[G]$-module

`GModuleAction(M)`

Given a $K[G]$-module M, return the action of G on M as homomorphism f of G into the matrix group $GL_n(K)$.

`Representation(M)`

Given a $K[G]$-module M, return the action of G on M as homomorphism f of G into the matrix algebra $M_n(K)$.

Example H90E8

The function `Representation` allows the easy calculation of group characters. We illustrate this with the 6-dimension module for the group A_7 constructed above.

```
> A7 := AlternatingGroup(7);
> M  := PermutationModule(A7, Vector(GF(11), [1,0,1,0,1,0,1]));
> phi := Representation(M);
> [ Trace(phi(c[3])) : c in Classes(A7) ];
[ 7, 3, 4, 1, 1, 2, 0, 0, 0 ]
```

Example H90E9

We present a procedure which, given a $K[G]$-module M, constructs its dual D.

```
> DualModule := function(M)
>       G := Group(M);
>       f := Representation(M);
>       return GModule(G, [ Transpose(f(G.i))^-1 : i in [1 .. Ngens(G)] ]);
> end function;
```

`ActionGenerator(M, i)`

`RightActionGenerator(M, i)`

The i-th generator of the (right) acting matrix algebra for the module M. That is, the image of the i-th group generator in the corresponding representation.

`ActionGenerators(M)`

Return the matrices giving the action on the module M as a sequence. These are the images of the generators of the group in the corresponding representation.

`NumberOfActionGenerators(M)`

`Nagens(M)`

The number of action generators (the number of generators of the algebra) for the $R[G]$-module M.

`ActionGroup(M)`

The matrix group generated by the action generators of M.

`Sections (G)`

Given a matrix group G defined over a finite field K, return the action of G on each composition factor of the natural K[G]-module for G.

Example H90E10

We construct the tensor square T of the natural module M of the matrix group $G = SL(3, 5)$ and then determine the action of G on each composition factor of T.

```
> G := SL(3, 5);
> M := GModule(G);
> T := TensorProduct(M, M);
> A := ActionGroup(T);
> S := Sections(A);
> #S;
2
```

There are just two composition factors of T, the symmetric square and the exterior square of M.

```
> S[2];
MatrixGroup(3, GF(5))
Generators:

    [1 0 0]
    [0 2 0]
    [0 0 3]

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

90.4 Standard Constructions

Given one or more existing modules, various standard constructions are available to construct new modules.

90.4.1 Changing the Coefficient Ring

In this collection of functions will be found utilities for changing the base ring of the module. Note that several of the functions for rewriting over a minimal field are restricted to rings $K[G]$ where K is a finite field.

`ChangeRing(M, S)`

Given an A-module M with base ring R, together with a ring S, such that there is a natural homomorphism from R to S, construct the module N with base ring S where N is obtained from M by coercing the components of the vectors of M into N. The corresponding homomorphism from M to N is returned as a second value.

`ChangeRing(M, S, f)`

Given a module M with base ring R, together with a ring S, and a homomorphism $f : R \to S$, construct the module N with base ring S, where N is obtained from M by applying f to the components of the vectors of M. The corresponding homomorphism from M to N is returned as a second value.

90.4.2 Writing a Module over a Smaller Field

The functions in this section currently only apply to $K[G]$-modules defined over a finite field K.

```
IsRealisableOverSmallerField(M)
```

Given a $K[G]$-module M, where K is a finite field, return true if M can be realised over a proper subfield F of K. The equivalent $F[G]$-module is also returned. The Glasby-Howlett algorithm is used to determine the smallest field over which M can be realised.

```
IsRealisableOverSubfield(M, F)
```

Let M be a $K[G]$-module, where K is a finite field of characteristic p, and let F be a finite field also of characteristic p. If it is possible to realise M over the subfield F of K, return true and the equivalent $F[G]$-module.

```
WriteOverSmallerField(M, F)
```

Given a module M of dimension d over a finite field E having degree e and a subfield F of E having degree f, write the action of M as $d*e/f$ by $d*e/f$ matrices over F and return the module and the isomorphism.

```
AbsoluteModuleOverMinimalField(M, F)
```

Let M be a $K[G]$-module, where K is a finite field of characteristic p, and let F be a finite field also of characteristic p. This function returns the module obtained by writing M over the smallest possible field containing F subject to the condition that the dimension of M does not increase. The Glasby-Howlett algorithm is used to determine the smallest field over which M can be realised.

```
AbsoluteModuleOverMinimalField(M)
```

Verbose	`Reduce`	*Maximum* : 2
Verbose	`Cohomology`	*Maximum* : 2
Verbose	`GrunwaldWang`	*Maximum* : 2

Let M be a $K[G]$-module, where K is a finite field of characteristic p or a number field. This function returns the module obtained by writing M over a field of smallest possible degree subject to the condition that the dimension of M does not increase. For modules over finite fields, a field of smallest degree is always a subfield of K, in this case, the Glasby-Howlet algorithm is used. For number fields, a different field might be necessary and a combination of Plesken's method and a constructive version of the Grunwald-Wang theorem is used.

`Minimize(R)`

`All`	BoolElt	*Default :* `false`
`Char`	AlgChtrElt	*Default :* `false`
`FindSmallest`	BoolElt	*Default :* `false`
Verbose	`Reduce`	*Maximum :* 2
Verbose	`Cohomology`	*Maximum :* 2
Verbose	`GrunwaldWang`	*Maximum :* 2

Let $R : G \to \mathrm{Gl}(n, K)$ be an absolutely irreducible representation over some number field K. This function tries to find minimal subfields k of K that afford R, ie. it tries to write the representation over a smaller field. In general however, there might be number field k not contained in K of smaller degree that afford R. If `All` is given, then instead of a single representation over a minimal degree subfield of K, a list of representations over all minimal subfields of K is returned instead. If `Char` is given, it should be set to the character of the representation. If `FindSmallest` is given, the field K will be extended by some auxiliary field A such that KA will contain a minimal degree field affording R. This involves a constructive version of the Grunwald-Wang theorem and can be computationally expensive if the degree of KA is too large.

`AbsoluteModulesOverMinimalField(Q, F)`

Let Q be a sequence of $K[G]$-modules, where K is a finite field of characteristic p, and let F be a finite field also of characteristic p. This function returns the sequence of modules obtained by writing each module M of Q over the smallest possible field containing F subject to the condition that the dimension of M does not increase. Thus, the effect of the function is to apply the function `AbsoluteModuleOverMinimalField` to each module of Q. The Glasby-Howlett algorithm is used to determine the smallest field over which the modules M of Q can be realised.

`ModuleOverSmallerField(M, F)`

Let M be a $K[G]$-module of dimension d, where K is a finite field of characteristic p, and let F be a subfield of K of index n. This function returns the $F[G]$-module N obtained by writing the action of M as $dn \times dn$ matrices over F.

`ModulesOverSmallerField(Q, F)`

Let Q be a sequence of $K[G]$-modules, where K is a finite field of characteristic p, and let F be a subfield of K of index n. This function returns the sequence R of $F[G]$-modules obtained by applying the function `ModuleOverSmallerField` to each term of Q. That is, each term N of R is formed by writing the action of the corresponding term of Q as $dn \times dn$ matrices over F.

`ModulesOverCommonField(M, N)`

Given $K[G]$-modules M and N, change their base fields to K, where K is the smallest field containing the base fields of M and N.

`WriteGModuleOver(M, K)`

`Char`	AlgChtrElt	*Default :* `false`
`Subfield`	BoolElt	*Default :* `false`
Verbose	`Reduce`	*Maximum :* 2
Verbose	`Cohomology`	*Maximum :* 2
Verbose	`GrunwaldWang`	*Maximum :* 2

Given a $L[G]$ module M and some number field K, try to write M over K. If `Char` is specified, it should be set to the character of this module. If `Subfield` is given, the module will be rewritten over a minimal degree subfield of K.

`WriteRepresentationOver(R, K)`

`Char`	AlgChtrElt	*Default :* `false`
`Subfield`	BoolElt	*Default :* `false`
Verbose	`Reduce`	*Maximum :* 2
Verbose	`Cohomology`	*Maximum :* 2
Verbose	`GrunwaldWang`	*Maximum :* 2

Given an absolutely irreducible representation $R : G \rightarrow \mathrm{Gl}(n, L)$ and some normal number field K, try to write R over K. If `Char` is specified, it should be set to the character of this representation. If `Subfield` is given, the representation will be rewritten over a minimal degree subfield of K.

Example H90E11

We will work with the G-module and character of the unique 2-dimensional character of Q_8. It is well known that, while the character is defined over Q, the corresponding representation can only be defined over fields where -1 is the sum of 2 squares.

```
> G := TransitiveGroup(8, 5);
> TransitiveGroupDescription(G);
Q_8(8);
> R := AbsolutelyIrreducibleModules(G, Rationals());
> R;
[
    GModule of dimension 1 over Rational Field,
    GModule of dimension 1 over Rational Field,
    GModule of dimension 1 over Rational Field,
    GModule of dimension 1 over Rational Field,
    GModule of dimension 2 over Cyclotomic Field
    of order 4 and degree 2
]
```

```
> R := R[5];
> WriteGModuleOver(R, CyclotomicField(5));
GModule of dimension 2 over Cyclotomic Field
of order 5 and degree 4
```

So $Q(\zeta_5)$ is an example of a field affording the module but having no minimal degree subfield (of degree 2 here) affording R!.

```
> AbsoluteModuleOverMinimalField($1);
GModule of dimension 2 over Number Field with
defining polynomial Qx.1^2 - Qx.1 + 1 over the
Rational Field
```

Note that the base field returned here is $Q(\zeta_3)$ which is of degree 2 but different from $Q(\zeta_4)$ that was found initially. In general there are infinetely many minimal degree splitting fields.
If we try to realize R over a field where -1 cannot be written as a sums of two squares we get an error:

```
>  WriteGModuleOver(R, QuadraticField(3));
>> WriteGModuleOver(R, QuadraticField(3));
                   ^
Runtime error in 'WriteGModuleOver': The G-module
cannot be realised over K
```

We can try to find a minimal field containing $Q(\sqrt{3})$ by computing the local Schur-indices and then obtain a splitting field:

```
> k := QuadraticField(3);
> SchurIndices(Character(R), k);
[ <1st place at infinity, 2>, <2nd place at
infinity, 2> ]
> A := SplittingField($1);
> A;
FldAb, defined by (<3>, [1      2])
of structure: Z/2
```

So the splitting field is returned as an abelian extension. We can see that A is of degree 2 over k and will be ramified at most at 3 and both infinite places. In order to use it to rewrite the module, we need to convert to a number field over Q first:

```
> A := NumberField(A);
> A;
Number Field with defining polynomial $.1^2 + 1
over k
> A := AbsoluteField(A);
> A;
Number Field with defining polynomial Qx.1^4 -
    4*Qx.1^2 + 16 over the Rational Field
> WriteGModuleOver(R, A);
GModule of dimension 2 over A
> WriteGModuleOver(R, A:Subfield);
```

```
GModule of dimension 2 over Number Field with
defining polynomial Qx.1^2 - Qx.1 + 1 over the
Rational Field
```

90.4.3 Direct Sum

`DirectSum(M, N)`

Given $K[G]$-modules M and N, construct the direct sum D of M and N as an $K[G]$-module. The embedding maps from M into D and from N into D respectively and the projection maps from D onto M and from D onto N respectively are also returned.

`DirectSum(Q)`

Given a sequence Q of $K[G]$-modules, construct the direct sum D of these modules. The embedding maps from each of the elements of Q into D and the projection maps from D onto each of the elements of Q are also returned.

90.4.4 Tensor Products of $K[G]$-Modules

`TensorProduct(M, N)`

Let M and N be two $K[G]$ modules. This function constructs the tensor product, $M \otimes_A N$, with diagonal action.

`TensorPower(M, n)`

Given a $K[G]$-module M and an integer $n \geq 1$, construct the n-th tensor power of M.

`ExteriorSquare(M)`

Given a $K[G]$-module M, construct the A-submodule of $M \otimes_A M$ consisting of the skew tensors.

`SymmetricSquare(M)`

Given a $K[G]$-module M, construct the A-submodule of $M \otimes_A M$ consisting of the symmetric tensors.

90.4.5 Induction and Restriction

```
Dual(M)
```

Given an $K[G]$-module M, construct the $K[G]$-module which is the K-dual, $\mathrm{Hom}_K(M, K)$, of M.

```
Induction(M, G)
```

Given a $K[H]$-module M and a supergroup G of H, construct the $K[G]$-module obtained by inducing M up to G.

```
Induction(R, G)
```

Given a representation R of a subgroup of G, construct the representation of G obtained by inducing R up to G.

```
Restriction(M, H)
```

Given a $K[G]$-module M and a subgroup H of G, form the $K[H]$-module corresponding to the restriction of M to the subgroup H.

Example H90E12

Starting with the permutation module M over $\mathbf{F}_2$ for the Mathieu group M_{22}, we apply the induction and restriction functions to find new irreducible modules for M_{22}.

```
> SetSeed(1);
> G := PermutationGroup< 22 |
>        (1,2,4,8,16,9,18,13,3,6,12)(5,10,20,17,11,22,21,19,15,7,14),
>        (1,18,4,2,6)(5,21,20,10,7)(8,16,13,9,12)(11,19,22,14,17),
>        (1,18,2,4)(3,15)(5,9)(7,16,21,8)(10,12,20,13)(11,17,22,14) >;
> M := PermutationModule(G, GaloisField(2));
> M;
GModule M of dimension 22 with base ring GF(2)
> CM := Constituents(M);
> CM;
[
    GModule of dimension 1 over GF(2),
    GModule of dimension 10 over GF(2),
    GModule of dimension 10 over GF(2)
]
```

We restrict the module M to the stabilizer of a point in M_{22} and then induce back up, a constituent of the restriction.

```
> L34 := Stabilizer(G, 1);
> N := Restriction(M, L34);
> N;
GModule N of dimension 22 with base ring GF(2)
> CN := Constituents(N);
> CN;
[
    GModule of dimension 1 over GF(2),
    GModule of dimension 9 over GF(2),
    GModule of dimension 9 over GF(2)
]
> Ind1 := Induction(CN[1], G);
> Ind1;
GModule Ind1 of dimension 22 over GF(2)
> Constituents(Ind1);
[
    GModule of dimension 1 over GF(2),
    GModule of dimension 10 over GF(2),
    GModule of dimension 10 over GF(2)
]
> Ind2 := Induction(CN[2], G);
> Ind2;
GModule Ind2 of dimension 198 over GF(2)
> Constituents(Ind2);
[
    GModule of dimension 1 over GF(2),
    GModule of dimension 10 over GF(2),
    GModule of dimension 10 over GF(2),
    GModule of dimension 34 over GF(2),
    GModule of dimension 98 over GF(2)
]
```

Thus, inducing up the 1-dimensional constituent of N gives us irreducible modules for G having the same dimensions as those appearing as constituents of M. However, inducing up the 9-dimensional module gives us irreducible modules of new dimensions: 34 and 98. Hence starting out with only the permutation module for M_{22} over $\mathbf{F}_2$, we have found 5 irreducible modules for the group.

90.4.6 The Fixed-point Space of a Module

```
Fix(M)
```

Given an $K[G]$-module M, construct the largest submodule of M on which G acts trivially, i.e. the fixed-point space of M.

90.4.7 Changing Basis

```
M ^ T
```

Given a $K[G]$-module M of dimension n over the field K, and a nonsingular $n \times n$ matrix T over K, construct the $K[G]$-module N which corresponds to taking the rows of T as a basis for M.

90.5 The Construction of all Irreducible Modules

The construction of all irreducible K[G]-modules for a finite group G is of major interest. If G is soluble there is a very effective method that dates back to Schur. This proceeds by working up a composition series for G and constructing the irreducibles for each subgroup by inducing or extending representations from the previous subgroup. This works equally well over finite fields and over fields of characteristic zero. If G is non-soluble the situation is more difficult. Starting with a faithful representation, by a theorem of Burnside, it is possible to construct all representations by splitting tensor powers of the faithful representation. An algorithm based on this idea developed by Cannon and Holt uses the Meataxe to split representations and works well over small finite fields. A similar algorithm for rational representations using a different method for splitting representations is under development by Steel. Methods based on the theorem of Burnside, will be referred to as *Burnside algorithms*.

90.5.1 Generic Functions for Finding Irreducible Modules

The functions described in this section construct all irreducible representations of a finite group. The choice of algorithm depends upon the type of group and the kind of field given. The individual algorithms may be invoked directly by means of intrinsic functions described in subsequent sections.

`IrreducibleModules(G, K : parameters)`

`AbsolutelyIrreducibleModules(G, K : parameters)`

Let G be a finite group and let K be a field. If G is soluble then K may be either a finite field, the rational field or a cyclotomic field whose order divides the exponent of G. If G is non-soluble, then currently K is restricted to being a finite field or the rational field. These functions construct all of the irreducible or absolutely irreducible G-modules over K. Since V2.16, if K is the rational field, then by default a new algorithm is used. Otherwise, if G is soluble, Schur's algorithm is normally used while if G is non-soluble, the Burnside method is used.

`Alg` MonStgElt *Default :*

The parameter `Alg` may be used to override the usual choice of algorithm if desired. Note that Schur's algorithm may only be applied to soluble groups, while, for the time being, Burnside's method requires that the field K is finite.

The Schur algorithm is usually very fast and is often able to find the complex representations more quickly than it is possible to compute the character table. The speed of the Burnside algorithm is determined firstly by the maximal degree of the irreducible modules and secondly by the number of irreducible modules. It will start to become quite slow if G has modules of dimension in excess of 1000. In order to prevent the Burnside method from wasting huge amounts of time, the algorithm takes a parameter which controls the degree of the largest module that will be consider for splitting.

`DimLim` RngInt *Default :* 2000

The parameter DimLim only affects the Burnside algorithm where it is used to limit the dimension of the modules which will be considered for splitting. If this limit prevents all irreducibles being found, a warning message is output and those irreducibles that have been found will be returned. This possibility allows the user to determine a sample of low degree modules without using excessive time.

Example H90E13

We take a group of order $416 = 2^5 13$ and compute its irreducible modules over various fields.

```
> G := SmallGroup(416, 136);
> G;
GrpPC : G of order 416 = 2^5 * 13
PC-Relations:
    G.1^2 = Id(G),
    G.2^2 = G.5,
    G.3^2 = Id(G),
    G.4^2 = G.5,
    G.5^2 = Id(G),
    G.6^13 = Id(G),
    G.3^G.1 = G.3 * G.5,
    G.3^G.2 = G.3 * G.4,
    G.4^G.2 = G.4 * G.5,
    G.4^G.3 = G.4 * G.5,
    G.6^G.1 = G.6^12
```

We first compute the $K[G]$-modules for the finite fields $K = GF(p)$, where p runs through the primes dividing the order of G.

```
> IrreducibleModules(G, GF(2));
[
    GModule of dimension 1 over GF(2),
    GModule of dimension 12 over GF(2)
]
> IrreducibleModules(G, GF(13));
[
    GModule of dimension 1 over GF(13),
    GModule of dimension 1 over GF(13),
    GModule of dimension 1 over GF(13),
    GModule of dimension 1 over GF(13),
    GModule of dimension 1 over GF(13),
    GModule of dimension 1 over GF(13),
    GModule of dimension 1 over GF(13),
    GModule of dimension 1 over GF(13),
    GModule of dimension 2 over GF(13),
    GModule of dimension 2 over GF(13),
    GModule of dimension 4 over GF(13)
]
```

We now compute the $K[G]$-modules where K is the rational field.

```
> time L := IrreducibleModules(G, Rationals());
Time: 16.170
> L;
[
    GModule of dimension 1 over Rational Field,
    GModule of dimension 1 over Rational Field,
    GModule of dimension 1 over Rational Field,
    GModule of dimension 1 over Rational Field,
    GModule of dimension 1 over Rational Field,
    GModule of dimension 1 over Rational Field,
    GModule of dimension 1 over Rational Field,
    GModule of dimension 1 over Rational Field,
    GModule of dimension 2 over Rational Field,
    GModule of dimension 2 over Rational Field,
    GModule of dimension 8 over Rational Field,
    GModule of dimension 12 over Rational Field,
    GModule of dimension 12 over Rational Field,
    GModule of dimension 12 over Rational Field,
    GModule of dimension 12 over Rational Field,
    GModule of dimension 24 over Rational Field,
    GModule of dimension 96 over Rational Field
]
```

Finally, we compute the $K[G]$-modules taking K to be the splitting field over the rationals and verify that the number of irreducibles is equal to the number of conjugacy classes of G.

```
> Exponent(G);
104
> mods := IrreducibleModules(G, CyclotomicField(104));
> #mods;
53;
> #Classes(G);
53
> [ Dimension(N) : N in mods ];
[ 1, 1, 1, 1, 1, 1, 1, 1, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
  2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 4, 4,
  4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4 ]
> X := CharacterTable(G);
> [ Degree(x) : x in X ];
[ 1, 1, 1, 1, 1, 1, 1, 1, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
  2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 4, 4,
  4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4 ]
```

90.5.2 The Burnside Algorithm

The Burnside algorithm finds all irreducible $K[G]$-modules with a faithful $K[G]$-module P and looks for the distinct irreducibles among the tensor powers of P. Both irreducible and absolutely irreducible modules may be found. At present the algorithm is restricted to finite fields. In more detail the algorithm starts with a some faithful permutation module over the given field, splits it into irreducibles using the meataxe, and constructing further modules to split as tensor products of those already found. A warning is printed if all irreducible modules are not found.

`AbsolutelyIrreducibleModulesBurnside(G, K : parameters)`

`DimLim` RngIntElt *Default :* 2000

Given a finite group G and a finite field K, this function constructs the absolutely irreducible $K[G]$-modules over extensions of K. Currently, the group G is restricted to a permutation group. The maximum dimension of a module considered for splitting is controlled by the parameter `DimLim`, which has default value 2000.

`IrreducibleModulesBurnside(G, K : parameters)`

`DimLim` RngIntElt *Default :* 2000

Given a finite group G and a finite field K, this function constructs the irreducible $K[G]$-modules over K. Currently, the group G is restricted to a permutation group. The maximum dimension of a module considered for splitting is controlled by the parameter `DimLim`, which has default value 2000.

`AbsolutelyIrreducibleConstituents(M)`

Given an irreducible module M, return M if it is already absolutely irreducible, else return the absolutely irreducible modules obtained by finding the constituents of M after extending the base field of M to a splitting field.

Example H90E14

We find all irreducible modules for M_{11} over $GF(2)$, and all absolutely irreducible modules of characteristic 2. The Burnside algorithm is used by default.

```
> load m11;
Loading "/home/magma/libs/pergps/m11"
M11 - Mathieu group on 11 letters - degree 11
Order 7 920 = 2^4 * 3^2 * 5 * 11;  Base 1,2,3,4
Group: G
> IrreducibleModules(G, GF(2));
[
  GModule of dimension 1 over GF(2),
  GModule of dimension 10 over GF(2),
  GModule of dimension 44 over GF(2),
  GModule of dimension 32 over GF(2)
]
> AbsolutelyIrreducibleModules(G, GF(2));
```

```
[
  GModule of dimension 1 over GF(2),
  GModule of dimension 10 over GF(2),
  GModule of dimension 44 over GF(2),
  GModule of dimension 16 over GF(2^2),
  GModule of dimension 16 over GF(2^2)
]
```

90.5.3 The Schur Algorithm for Soluble Groups

This collection of functions allows the user to find all irreducible modules of a finite soluble group. The group is first replaced by an isomorphic group defined by a power-conjugate presentation. The irreducibles are then found using Schur's method of working up the composition series for G defined by the pc-presentation.

`IrreducibleModules(G, K :` *parameters*`)`

`AbsolutelyIrreducibleModulesSchur(G, K:` *parameters*`)`

Let G be a finite soluble group and let K be one the following types of field: a finite field, the rational field or a cyclotomic field. The order of a cyclotomic field must divide the exponent of G. The function constructs all absolutely irreducible representations of G over appropriate extensions or subfields of the field K. The modules returned are non-isomorphic and consist of all distinct modules, subject to the conditions imposed. In the case when K is a finite field, the Glasby-Howlett algorithm is used to determine the minimal field over which an irreducible module may be realised. If K has characteristic 0, the field over which an irreducible module is given may not be minimal.

The irreducible modules are found using Schur's method of climbing the composition series for G defined by the pc-presentation.

`Process` BoolElt *Default :* `true`

If the parameter `Process` is set true then the list is a list of pairs comprising an integer and a representation. This list or any sublist of it is a suitable value for the argument L in the last versions of the function, and in this case only the representations in L will be extended up the series. This allows the user to inspect the representations produced along the way and cull any that are uninteresting.

`GaloisAction` MonStgElt *Default :* "Yes"

Possible values are "Yes", "No" and "Relative". The default is "Yes" for intermediate levels and "No" for the whole group. The value "Yes" means that it only lists one irreducible from each orbit of the action of the absolute Galois group $Gal(K/\text{primefield}(K))$. Setting this parameter to "No" turns this reduction off (thus listing all inequivalent representations), while setting it to "Relative" uses the group $Gal(K/k)$.

`MaxDimension` RngIntElt *Default :*

Restrict the irreducible to those of dimension ≤ MaxDimension. Default is no restriction.

`ExactDimension` SetEnum *Default :*

If `ExactDimension` is assigned a set S of positive integers, attention is restricted to irreducible having dimensions lying in the set S. The default is equivalent to taking the set of all positive integers.

If both `MaxDimension` and `ExactDimension` are assigned values, then irreducible having dimensions that are either bounded by `MaxDimension` or contained in `ExactDimension` are produced.

`IrreducibleModulesSchur(G, K: parameters)`

Compute irreducible modules for G over the given field K. All arguments and parameters are as for the absolutely irreducible case.

The computation proceeds by first computing the absolutely irreducible representations subject to the given conditions, then rewriting over the field K, with a consequent change of dimension of the representation.

Example H90E15

We compute representations of the dihedral group of order 20.

```
> G := DihedralGroup(GrpPC, 10);
> FactoredOrder(G);
[ <2, 2>, <5, 1> ]
```

First some modular representations with characteristic 2.

```
> r := IrreducibleModulesSchur(G, GF(2));
> r;
[*
    GModule of dimension 1 over GF(2),
    GModule of dimension 4 over GF(2)
*]
> r := AbsolutelyIrreducibleModulesSchur(G, GF(2));
> r;
[*
    GModule of dimension 1 over GF(2),
    GModule of dimension 2 over GF(2^2),
    GModule of dimension 2 over GF(2^2)
*]
> r := AbsolutelyIrreducibleModulesSchur(G, GF(2) : GaloisAction:="Yes");
> r;
[*
    GModule of dimension 1 over GF(2),
    GModule of dimension 2 over GF(2^2)
*]
```

The irreducible representation of dimension 4 is not absolutely irreducible, as over $GF(4)$ it splits into two Galois-equivalent representations.

Finding irreducible modules over the complex field is straightforward, despite not being able to use the complex field as the field argument. We could instead specify the cyclotomic field having order equal to the exponent of G, but it is preferable to ask for all absolutely irreducible modules over the rationals.

```
> r := AbsolutelyIrreducibleModulesSchur(G, Rationals());
> r;
[*
   GModule of dimension 1 over Rational Field,
   GModule of dimension 1 over Rational Field,
   GModule of dimension 1 over Rational Field,
   GModule of dimension 1 over Rational Field,
   GModule of dimension 2 over Cyclotomic Field of order 5 and degree 4,
   GModule of dimension 2 over Cyclotomic Field of order 5 and degree 4,
   GModule of dimension 2 over Cyclotomic Field of order 5 and degree 4,
   GModule of dimension 2 over Cyclotomic Field of order 5 and degree 4
*]
> Representation(r[6])(G.2);
[zeta_5^3       0]
[       0 zeta_5^2]
```

```
AbsolutelyIrreducibleRepresentationsInit(G, F : parameters)
AbsolutelyIrreducibleModulesInit(G, F : parameters)
IrreducibleRepresentationsInit(G, F : parameters)
IrreducibleModulesInit(G, F : parameters)
```

Initialize a Process for calculating all linear representations of a soluble group G over the field F. The field F is restricted to either a finite field or the rationals. The parameters are as described above.

```
NextRepresentation(P)
NextModule(P)
```

Return `true` and the next representation from the process P, if there is one, or just `false` if the process is exhausted.

```
AbsolutelyIrreducibleRepresentationProcessDelete(~P)
```

Free all data associated with the process P.

90.5.4 The Rational Algorithm

Magma V2.16 contains a new algorithm for constructing irreducible rational representations of an arbitrary finite group. This algorithm is now selected by default in the function `IrreducibleModules(G, K)` when the field K is $\mathbf{Q}$.

Given an irreducible complex character for a group G, the sum of its Galois orbit gives an *irreducible rational character* for G. The Schur index of an irreducible rational character χ is defined to be the Schur index of an irreducible complex character whose Galois orbit sum is χ.

We call the sequence of all possible irreducible rational characters for G (sorted by degree) the *rational character table* of G. The irreducible modules returned by `IrreducibleModules(G, RationalField())` always match the rational character table.

`IrreducibleModules(G, Q : parameters)`

Given a finite group G, this function returns a sequence L of all irreducible rational $K[G]$-modules, and also the rational character table of G as a sequence C, which matches L. The character of the i-th module $L[i]$ is always $s_i \cdot C[i]$, where s_i is the Schur index of $C[i]$. Thus the dimension of the irreducible module $L[i]$ is $s_i{\cdot}\mathrm{Deg}(C[i])$.

`MaxDegree` RngIntElt *Default :* 0

Setting the parameter `MaxDegree` to a positive integer D instructs the algorithm not to spend effort on constructing irreducible modules whose corresponding irreducible rational characters have degree greater than D. Note that irreducible modules whose character degrees greater than D may be returned in any case, if they are easily constructed (often as side effects of operations used to construct smaller-dimensional modules).

`Characters` [AlgChtrElt] *Default :*

Setting the parameter `Characters` to a set or sequence S of characters for the group G instructs the algorithm to attempt to construct only the irreducible modules whose characters are in S. Each character in S may either be an irreducible rational character, or a (complex) ordinary character χ, in which case the irreducible rational character corresponding to χ (the orbit sum of χ) is used. As for the parameter `MaxDegree`, irreducible modules whose characters are not in S may be returned in any case, if they are easily constructed or are needed as intermediate modules to construct the desired modules.

`CharacterDegrees` [RngIntElt] *Default :*

Setting the parameter `CharacterDegrees` to a set or sequence I of positive integers instructs the algorithm to attempt to construct only the irreducible modules whose character degrees are in S. This is equivalent to setting the parameter `Characters` to `[c: c in RationalCharacterTable(G) | Degree(c) in I]`.

`UseInduction` BoolElt *Default :* `true`

By default, as the algorithm proceeds, it automatically searches for subgroups H_i of G such that irreducible rational H_i-modules may be induced up to G to yield

G-modules from which irreducible *G*-modules may be computed. In general, this method is very effective (and often yields modules for *G* with small entries) but can be very slow for some groups (particularly when *G* has many subgroups). Thus setting the parameter `UseInduction` to `false` will force the algorithm not to use induction.

```
RationalCharacterTable(G)
```

Given a finite group G, return the rational character table of G as a sequence C of the irreducible rational characters of G (sorted by degree), and also a index sequence I, such that for each i, $C[i]$ is the sum of the Galois orbit of $T[I[i]]$, where T is the ordinary (complex) character table of G.

Example H90E16

We compute all irreducible rational modules for PSL$(3,3)$ and note that the characters of the resulting modules match the entries in the rational character table.

```
> G := PSL(3, 3); #G;
5616
> T := CharacterTable(G); [Degree(x): x in T];
[ 1, 12, 13, 16, 16, 16, 16, 26, 26, 26, 27, 39 ]
> C := RationalCharacterTable(G); C;
[
    ( 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1 ),
    ( 12, 4, 3, 0, 0, 1, 0, 0, -1, -1, -1, -1 ),
    ( 13, -3, 4, 1, 1, 0, -1, -1, 0, 0, 0, 0 ),
    ( 26, 2, -1, -1, 2, -1, 0, 0, 0, 0, 0, 0 ),
    ( 27, 3, 0, 0, -1, 0, -1, -1, 1, 1, 1, 1 ),
    ( 39, -1, 3, 0, -1, -1, 1, 1, 0, 0, 0, 0 ),
    ( 52, -4, -2, -2, 0, 2, 0, 0, 0, 0, 0, 0 ),
    ( 64, 0, -8, 4, 0, 0, 0, 0, -1, -1, -1, -1 )
]
> time L := IrreducibleModules(G, RationalField());
Time: 0.760
> L;
[
    GModule of dimension 1 over Rational Field,
    GModule of dimension 12 over Rational Field,
    GModule of dimension 13 over Rational Field,
    GModule of dimension 26 over Rational Field,
    GModule of dimension 27 over Rational Field,
    GModule of dimension 39 over Rational Field,
    GModule of dimension 52 over Rational Field,
    GModule of dimension 64 over Rational Field
]
> [Character(M): M in L];
[
    ( 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1 ),
    ( 12, 4, 3, 0, 0, 1, 0, 0, -1, -1, -1, -1 ),
    ( 13, -3, 4, 1, 1, 0, -1, -1, 0, 0, 0, 0 ),
    ( 26, 2, -1, -1, 2, -1, 0, 0, 0, 0, 0, 0 ),
    ( 27, 3, 0, 0, -1, 0, -1, -1, 1, 1, 1, 1 ),
    ( 39, -1, 3, 0, -1, -1, 1, 1, 0, 0, 0, 0 ),
    ( 52, -4, -2, -2, 0, 2, 0, 0, 0, 0, 0, 0 ),
    ( 64, 0, -8, 4, 0, 0, 0, 0, -1, -1, -1, -1 )
]
```

Example H90E17

For large groups, one can use the parameter `MaxDegree` to compute the irreducible modules of reasonable dimension.

```
> load m23;
Loading "/home/magma/libs/pergps/m23"
M23 - Mathieu group on 23 letters - degree 23
Order 10 200 960 = 2^7 * 3^2 * 5 * 7 * 11 * 23;  Base 1,2,3,4,5,6
Group: G
> T := CharacterTable(G); [Degree(x): x in T];
[ 1, 22, 45, 45, 230, 231, 231, 231, 253, 770, 770,
896, 896, 990, 990, 1035, 2024 ]
> C := RationalCharacterTable(G); C;
[
    ( 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1 ),
    ( 22, 6, 4, 2, 2, 0, 1, 1, 0, 0, 0, -1, -1, -1, -1, -1, -1 ),
    ( 90, -6, 0, 2, 0, 0, -1, -1, -2, 2, 2, 1, 1, 0, 0, -2, -2 ),
    ( 230, 22, 5, 2, 0, 1, -1, -1, 0, -1, -1, 1, 1, 0, 0, 0, 0 ),
    ( 231, 7, 6, -1, 1, -2, 0, 0, -1, 0, 0, 0, 0, 1, 1, 1, 1 ),
    ( 253, 13, 1, 1, -2, 1, 1, 1, -1, 0, 0, -1, -1, 1, 1, 0, 0 ),
    ( 462, 14, -6, -2, 2, 2, 0, 0, -2, 0, 0, 0, 0, -1, -1, 2, 2 ),
    ( 1035, 27, 0, -1, 0, 0, -1, -1, 1, 1, 1, -1, -1, 0, 0, 0, 0 ),
    ( 1540, -28, 10, -4, 0, 2, 0, 0, 0, 0, 0, 0, 0, 0, 0, -1, -1 ),
    ( 1792, 0, -8, 0, 2, 0, 0, 0, 0, -1, -1, 0, 0, 2, 2, -2, -2 ),
    ( 1980, -36, 0, 4, 0, 0, -1, -1, 0, 0, 0, -1, -1, 0, 0, 2, 2 ),
    ( 2024, 8, -1, 0, -1, -1, 1, 1, 0, 0, 0, 1, 1, -1, -1, 0, 0 )
]
> Q := RationalField();
> time L := IrreducibleModules(G, Q: MaxDegree := 253);
Time: 23.400
> L;
[
    GModule of dimension 1 over Rational Field,
    GModule of dimension 22 over Rational Field,
    GModule of dimension 90 over Rational Field,
    GModule of dimension 230 over Rational Field,
    GModule of dimension 231 over Rational Field,
    GModule of dimension 253 over Rational Field,
    undef,
    undef,
    undef,
    undef,
    GModule of dimension 1980 over Rational Field
]
```

Note that the module of dimension 1980 is included (it was easily constructed as the tensor product of modules of dimension 20 and 90).

90.6 Extensions of Modules

For $K[G]$-modules M and N, the K-vector $\mathrm{Ext}(M, N)$ of equivalence classes of $K[G]$-module extensions

$$0 \to N \to L \to M \to 0$$

of N by M can be computed, and corresponding extensions L constructed.

```
Ext(M, N)
```

Given $K[G]$-modules M and N, construct the K-vector space $\mathrm{Ext}(M, N)$ of equivalence classes of $K[G]$-module extensions of N by M.

```
Extension(M, N, e)
```

Construct a $K[G]$-module extension L of N by M corresponding to the element e of E, where E must be the vector space returned by a previous call of `Ext(M,N)`. The insertion $N \to L$ and projection $L \to M$ are also returned.

```
MaximalExtension(M, N, E)
```

Again E must be the vector space returned by a previous call of `Ext(M,N)`. Construct the largest possible $K[G]$-module extension L of a direct sum of copies of N by M, such that none of the submodules of L that are isomorphic to N has a complement in L.

Example H90E18

```
> G := Alt(5);
> I := IrreducibleModules(G, GF(2));
> I;
[
    GModule of dimension 1 over GF(2),
    GModule of dimension 4 over GF(2),
    GModule of dimension 4 over GF(2)
]
> M1 := rep{M: M in I | Dimension(M) eq 1};
> M4 := rep{M: M in I | Dimension(M) eq 4 and not IsAbsolutelyIrreducible(M)};
> M4; assert not IsAbsolutelyIrreducible(M4);
```

```
GModule M of dimension 4 over GF(2)
> E, rho := Ext(M4, M1);
> E;
Full Vector space of degree 2 over GF(2)
> Extension(M4, M1, E.1, rho);
GModule of dimension 5 over GF(2)
[0 0 0 0 1]
[1 0 0 0]
[0 1 0 0]
[0 0 1 0]
[0 0 0 1]
[0 0 0 0]
> E := MaximalExtension(M4, M1, E, rho);
> E;
GModule E of dimension 6 over GF(2)
> CompositionFactors(E);
[
    GModule of dimension 1 over GF(2),
    GModule of dimension 1 over GF(2),
    GModule of dimension 4 over GF(2)
]
```

90.7 The Construction of Projective Indecomposable Modules

For a finite group G and a finite field K, the projective indecomposable $K[G]$-modules are in one-one correspondence with the irreducible $K[G]$-modules, where the projective indecomposable module P corresponding to the irreducible module I has the property that Socle(P) and P/JacobsonRadical(P) are both isomorphic to I.

Magma functions for the construction of the irreducible modules were described in Subsection 90.5.1. The functions described in this section may be used to construct the corresponding projective indecomposables for finite groups of moderate order – up to around 10^6, depending on the example. Large-dimensional projective indecomposable modules can sometimes be constructed by using *condensation* techniques, which allow many of the necessary computations to be carried out in condensed modules, which may have significantly smaller dimension than their standard uncondensed equivalents. As with the computation of irreducible modules, these methods work best for permutation groups and PC-groups.

The verbose flag "KGModule" may be set to 1 or 2 to show details of the computations.

```
ProjectiveIndecomposableDimensions(G, K)
```

The K-dimensions of the projective indecomposable $K[G]$-modules corresponding to the irreducible $K[G]$-modules returned by `IrreducibleModules(G,K)`. (These can be computed quickly from the Brauer characters of the irreducible modules, using the Cartan matrix discussed below.)

`ProjectiveIndecomposableModule(I:` *parameters*`)`

Construct and return the projective indecomposable $K[G]$-module P corresponding to the irreducible $K[G]$-module I. Note that Socle(P) and P/JacobsonRadical(P) are both isomorphic to I.

`condensation` BoolElt *Default :* `false`

If set to `true`, then an attempt is made to find a subgroup of G which allows computations to be carried out in condensed versions of the modules involved.

```
ProjectiveIndecomposableModules(G, K)
```

Construct the complete list of projective indecomposable $K[G]$-modules corresponding to the irreducible $K[G]$-modules returned by `IrreducibleModules(G,K)`.

`condensation` BoolElt *Default :* `false`

If set to `true`, then an attempt is made to find a subgroup of G which allows computations to be carried out in condensed versions of the modules involved.

Example H90E19

We compute the projective indecomposable modules for the alternating group of degree 8 over the field of order 2.

```
> G := Alt(8);
> K := GF(2);
> IrreducibleModules(G, K);
[
    GModule of dimension 1 over GF(2),
    GModule of dimension 4 over GF(2),
    GModule of dimension 4 over GF(2),
    GModule of dimension 6 over GF(2),
    GModule of dimension 14 over GF(2),
    GModule of dimension 20 over GF(2),
    GModule of dimension 20 over GF(2),
    GModule of dimension 64 over GF(2)
]
> ProjectiveIndecomposableDimensions(G, K);
[ 448, 192, 192, 320, 320, 192, 192, 64 ]
> time proj := ProjectiveIndecomposables(G, K);
Time: 22.070
> proj;
[
    GModule of dimension 448 over GF(2),
    GModule of dimension 192 over GF(2),
    GModule of dimension 192 over GF(2),
    GModule of dimension 320 over GF(2),
    GModule of dimension 320 over GF(2),
    GModule of dimension 192 over GF(2),
    GModule of dimension 192 over GF(2),
    GModule of dimension 64 over GF(2)
]
> CompositionFactors(proj[1]);
[
    GModule of dimension 1 over GF(2),
    GModule of dimension 20 over GF(2),
    GModule of dimension 20 over GF(2),
    GModule of dimension 1 over GF(2),
    GModule of dimension 14 over GF(2),
    GModule of dimension 6 over GF(2),
    GModule of dimension 1 over GF(2),
    GModule of dimension 20 over GF(2),
    GModule of dimension 20 over GF(2),
    GModule of dimension 1 over GF(2),
    GModule of dimension 14 over GF(2),
    GModule of dimension 4 over GF(2),
    GModule of dimension 4 over GF(2),
    GModule of dimension 14 over GF(2),
    GModule of dimension 6 over GF(2),
    GModule of dimension 1 over GF(2),
    GModule of dimension 4 over GF(2),
    GModule of dimension 4 over GF(2),
    GModule of dimension 20 over GF(2),
    GModule of dimension 20 over GF(2),
    GModule of dimension 1 over GF(2),
    GModule of dimension 14 over GF(2),
    GModule of dimension 6 over GF(2),
    GModule of dimension 1 over GF(2),
    GModule of dimension 6 over GF(2),
    GModule of dimension 14 over GF(2),
    GModule of dimension 6 over GF(2),
    GModule of dimension 1 over GF(2),
    GModule of dimension 20 over GF(2),
    GModule of dimension 4 over GF(2),
    GModule of dimension 20 over GF(2),
    GModule of dimension 1 over GF(2),
    GModule of dimension 1 over GF(2),
    GModule of dimension 1 over GF(2),
    GModule of dimension 14 over GF(2),
    GModule of dimension 1 over GF(2),
    GModule of dimension 20 over GF(2),
    GModule of dimension 1 over GF(2),
    GModule of dimension 6 over GF(2),
    GModule of dimension 4 over GF(2),
    GModule of dimension 20 over GF(2),
    GModule of dimension 14 over GF(2),
    GModule of dimension 4 over GF(2),
    GModule of dimension 4 over GF(2),
    GModule of dimension 1 over GF(2),
    GModule of dimension 20 over GF(2),
    GModule of dimension 6 over GF(2),
    GModule of dimension 20 over GF(2),
    GModule of dimension 1 over GF(2),
    GModule of dimension 6 over GF(2),
    GModule of dimension 14 over GF(2),
    GModule of dimension 1 over GF(2)
]
```

`CartanMatrix(G, K)`

Let k be the number of irreducible $K[G]$-modules. The Cartan Matrix C for G over K is a $k \times k$ matrix of integers, in which the entry C_{ij} is equal to the number of times that the j-th irreducible $K[G]$-module is a constituent of the i-th projective indecomposable $K[G]$-module. This can be computed quickly from the Brauer characters of the irreducible $K[G]$-modules, and is used in `ProjectiveIndecomposableDimensions`.

(Note that, unlike the absolute Cartan matrix discussed below, C need not be symmetric.)

`AbsoluteCartanMatrix(G, K)`

This is the Cartan matrix of G over an extension L of K that is large enough for all irreducible $L[G]$-modules to be absolutely irreducible. Its rows and columns correspond to the $K[G]$-modules returned by `AbsolutelyIrreducibleModules(G,K)`.

It is a symmetric matrix with integer entries, and is equal to the Cartan matrix for G in the characteristic p of K, as defined in textbooks on modular representation theory.

`DecompositionMatrix(G, K)`

The decomposition matrix D of G in the characteristic p of K, as defined in textbooks on modular representation theory. The entry D_{ij} is equal to the number of times that the j-th absolutely irreducible $K[G]$-module occurs as a constituent of the i-th ordinary irreducible G-module over the complex numbers reduced modulo p. Note that $D^{\mathrm{T}}D$ is equal to the absolute Cartan matrix.

```
ProjectiveCover(M)
```

Compute the projective cover P of $K[G]$-module M together with a $K[G]$-module epimorphism $P \to M$ returned as a matrix.

If P_i is the projective indecomposable $K[G]$-module corresponding to the irreducible $K[G]$-module I_i and $M/\text{JacobsonRadical}(M)$ is isomorphic to $\oplus_{j=1}^{t} I_{i_j}$ then P is isomorphic to $\oplus_{j=1}^{t} P_{i_j}$.

```
CohomologicalDimension(M, n)
```

For $K[G]$-module M (with K a finite field and G a finite group), compute and return the K-dimension of the cohomology group $H^n(G, M)$ for $n \geq 0$. For $n = 0$ and 1, this is carried out by using the functions described in Chapter 68. For $n \geq 2$, it is done recursively using projective covers and dimension shifting to reduce to the case $n = 1$. (In particular, for $n = 2$, the method is different from that employed by the corresponding function for a cohomology module described in Chapter 68.)

```
CohomologicalDimensions(M, n)
```

For $K[G]$-module M (with K a finite field and G a finite group), compute and return the sequence of K-dimensions of the cohomology groups $H^k(G, M)$ for $1 \leq k \leq n$. On account of the recursive method used, this is quicker than computing them individually.

Example H90E20

We compute the cohomology of the irreducible modules for Alt(8) over the field of order 2 computed in the previous example.

```
> G := Alt(8);
> K := GF(2);
> irr :=  IrreducibleModules(G, K);
> [ CohomologicalDimension(I, 1) : I in irr ];
[ 0, 0, 0, 1, 1, 1, 1, 0 ]
> time [ CohomologicalDimension(I, 2) : I in irr ];
[ 1, 1, 1, 0, 2, 0, 0, 0 ]
Time: 0.440
> time [ CohomologicalDimension(I, 3) : I in irr ];
[ 2, 1, 1, 1, 1, 1, 1, 0 ]
Time: 15.070
> time [ CohomologicalDimension(I, 4) : I in irr ];
[ 2, 1, 1, 2, 3, 2, 2, 0 ]
Time: 99.500
> time CohomologicalDimensions(irr[1], 6);
[ 0, 1, 2, 2, 3, 6 ]
Time: 139.270
```

91 CHARACTERS OF FINITE GROUPS

Chapter 91
CHARACTERS OF FINITE GROUPS

Assume that G is a finite group of exponent m with k conjugacy classes of elements. The operators discussed here are concerned with the ring of *class functions on G*, defined to be the ring of complex-valued functions on G that are constant on conjugacy classes. This ring is made into a $\mathbf{C}$-algebra by identifying $c \in \mathbf{C}$ with the constant function that is c everywhere. In fact we will restrict ourselves to functions with values that are elements of cyclotomic fields.

Elements of the ring, ie. objects of type `AlgChtrElt`, are represented by the k values (elements of some cyclotomic field $\mathbf{Q}(\zeta_n)$) on the classes. The numbering of those elements matches the numbering of the classes as returned by `Classes` applied to the underlying group `G`: Thus `X[i]` is the value of the character `X` on the i-th class, ie. `Classes(G)[i]`.

91.1 Creation Functions

91.1.1 Structure Creation

```
ClassFunctionSpace(G)
```

```
CharacterRing(G)
```

Given a finite group G, create the ring of complex-valued class functions. This function will trigger the computation of the conjugacy classes of G if these are not yet known. Information about the irreducible characters is stored in the ring when it is computed.

91.1.2 Element Creation

The elementary constructions for class functions are listed. Other useful ways of defining class functions and characters are defined in sections discussing the permutation character, the (de)composition functions, and the sections on the conjugating, restricting and inducing of class functions.

`elt<  R | ` a_1`, ..., `a_k *:parameters* `>`

`R ! [ `a_1`, ..., `a_k` ]`

Given the ring of class functions R of a finite group G with k conjugacy classes and k elements a_i contained in some common cyclotomic field, create a class function on G for which the value on the i-th class is equal to the i-th term a_i.

`Character`   BoolElt   *Default :* `false`

If `Character := true`, then the resulting character is flagged to be a proper character.

```
R ! a
```

Define a constant class function for the ring of class functions R of the group G. Here a is allowed to be an integer, a rational field element or a cyclotomic field element.

```
Id(R)
Identity(R)
One(R)
PrincipalCharacter(G)
```

Given the finite group G or its ring of class functions R, create the principal character (which takes on the value 1 on every element of G).

```
Zero(R)
```

Given a ring of class functions R create its zero element (which is the class function that takes on the value 0 on every element of the group).

91.1.3 The Table of Irreducible Characters

The function `CharacterTable` can be invoked to determine the complete or a partial table of irreducible characters on a finite group G. If necessary, an existing character table can be supplemented by another call to the function. The known irreducible characters are stored in the character ring of G.

The default algorithm for character tables is Unger's Induce/Reduce algorithm, described in detail in [Ung06]. Computing character tables assumes that the classes of the group can be computed and stored, along with a class representative for each class, as well as the power map of the group. Unger's algorithm constructs elementary subgroups of the group (where elementary means direct product of cyclic group with p-group), and constructs characters of these subgroups (using Conlon's algorithm for the character table of the p-part of the group). Following Brauer's theorem on induced characters, these characters are induced to the full group and all irreducible characters of the full group are found in the integral span of the induced characters (using LLL reduction to maintain a manageable basis of the character space found). The two potentially most time-consuming parts of the algorithm are the computation of fusion in the elementary subgroups, which is necessary for induction, and repeated use of LLL to maintain a basis of the space of generalised characters found to date. Arithmetic with generalised characters is done quickly by using a finite field representation of these characters, as in Dixon's work [Dix67], but here the prime used may be twice the length of Dixon's.

The Dixon-Schneider algorithm for computing character tables [Dix67, Sch90] is also available. See below for details on how to access it. This was the previous default algorithm for character tables. As indicated above, Conlon's algorithm for the character table of a p-group [Con90a] is also implemented, and may be accessed directly as indicated below.

The functions in this section return character tables, which are enumerated sequences of characters that are flagged to allow printing in a special format.

`KnownIrreducibles(G)`

Given a finite group G, or a class function space R of G, return the table of irreducible characters currently stored. Such a table is a sequence of characters with specially formatted printing.

It should be noted that characters are stored with some information about whether they are characters, generalized characters or class functions. From this information, it is often possible to deduce with little effort that a character is, in fact, irreducible. When a new irreducible is found this way, it is immediately inserted into the table of irreducible characters.

`CharacterTable(G :`*parameters*`)`

Construct the table of ordinary irreducible characters for the group G.

`Al`	MonStgElt	*Default :* "*Default*"

This parameter controls the algorithm used. The string `"DS"` forces use of the Dixon-Schneider algorithm. The string `"IR"` forces the use of Unger's Induce/Reduce algorithm [Ung06]. The string `"Conlon"` forces the use of Conlon's algorithm. This last is only valid when the group is a p-group. The `"Default"` algorithm is to use Dixon-Schneider for groups of order ≤ 5000, Conlon's algorithm for larger p-groups and Unger's algorithm for other groups. This may change in future.

`DSSizeLimit`	RngIntElt	*Default :* 0

When the algorithm selected is either "Default" or "IR", a positive value n for `DSSizeLimit` means that before using ithe Induce/Reduce algorithm, the full possible character space is split by some passes of the Dixon-Schneider algorithm, restricted to using class matrices corresponding to conjugacy classes with size at most n.

`CharacterTableDS(G :`*parameters*`)`

`ClassMatrices`	SeqEnum	*Default :* []
`ClassMatrixLimit`	RngIntElt	*Default :* ∞
`ClassSizeLimit`	RngIntElt	*Default :* ∞
`MinChars`	RngIntElt	*Default :* ∞
`Modulus`	RngIntElt	*Default :* 0

Given a finite group G, construct the table of irreducible characters of G using the Dixon-Schneider algorithm [Dix67, Sch90]. The conjugacy classes of G will be computed if necessary; if the user wishes to exert control over the computation of the classes, the function `Classes` can be invoked on G before calling `CharacterTableDS`.

There are several optional parameters for `CharacterTableDS`. Assigning a non-negative integer value m to `MinChars`, indicates that the computation is to be terminated as soon as m or more additional characters have been found. By setting `ClassMatrixLimit` to a positive integer n, the user can limit the number of class matrices used. If the calculation of all irreducible characters (or at least m of them if MinChars has been set) cannot be completed by using k class matrices, an incomplete character table is returned. Setting ClassSizeLimit to an integer n restricts the algorithm to computing class matrices corresponding to classes of size at most n. Again, if the calculation of all irreducible characters cannot be completed using class matrices from these small classes, an incomplete character table is returned. The optional argument ClassMatrices (a sequence of integers in the range $1 \dots k$, where k is the number of conjugacy classes of G) can be used to specify a preference for the order in which class matrices should be used. Finally the Modulus argument can be used to set which prime field is used for the internal computations of the Dixon-Schneider algorithm. The value used must be a prime equivalent to 1 modulo the group exponent and greater than twice the square root of the group order.

The second return argument is the sequence of unsplit character spaces remaining at the point the algorithm terminated. This is empty when the algorithm returns a complete character table, but may be useful when an incomplete character table is returned.

```
Basis(R)
```

The function Basis takes the character ring R of G as input and performs the same computation to find the basis for R consisting of the irreducible characters.

Both CharacterTable and Basis return a character table, which is an enumerated sequence of elements of the character ring over G (if necessary, the ring is created) that only differs from arbitrary sequences of class functions with respect to printing.

```
CharacterTableConlon(G)
```

Compute the character table of group G using Conlon's algorithm for p-groups. The group group G must thus be a p-group for some prime p.

```
LinearCharacters(G)
```

Given a finite group G, determine the (partial) character table containing only the linear characters.

```
CharacterDegrees(G)
```

```
CharacterDegrees(G)
```

```
CharacterDegrees(G)
```

```
CharacterDegrees(G)
```

Returns the degrees of the ordinary irreducible characters of the finite group G. The sequence returned has form $[< d_1, c_1 >, < d_2, c_2 >, \dots]$ where c_i is the number of characters of degree d_i. For p-groups Slattery's algorithm is used, for other soluble groups Conlon's counting algorithm is used, and for insoluble groups the character table of G is computed.

`CharacterDegrees(G, z, p)`

Returns the degrees of the absolutely irreducible characters of G lying over a faithful linear character of $\langle z\rangle$, where z is a central element of G and p is zero or a prime.

`CharacterDegreesPGroup(G)`

Returns the degrees of the ordinary irreducible characters of the finite p-group G. The sequence returned has form $[c_0, c_1, c_2 \ldots]$, where c_i is the number of characters of degree p^i. Slattery's counting algorithm is used.

`RationalCharacterTable(G)`

Returns a sequence of minimal rational characters of G. These are the sums of the Galois orbits on the character table of G.

Example H91E1

We use the `CharacterTable` function and print the resulting table. Character tables have a special print format.

```
> G := Alt(5);
> CT := CharacterTable(G);
> CT;


Character Table of Group G
--------------------------



---------------------------
Class |   1  2  3    4    5
Size  |   1 15 20   12   12
Order |   1  2  3    5    5
---------------------------
p  =  2   1  1  3    5    4
p  =  3   1  2  1    5    4
p  =  5   1  2  3    1    1
---------------------------
X.1   +   1  1  1    1    1
X.2   +   3 -1  0   Z1 Z1#2
X.3   +   3 -1  0 Z1#2   Z1
X.4   +   4  0  1   -1   -1
X.5   +   5  1 -1    0    0


Explanation of Character Value Symbols
--------------------------------------


# denotes algebraic conjugation, that is,
#k indicates replacing the root of unity w by w^k

Z1     = (CyclotomicField(5: Sparse := true)) ! [
RationalField() | 1, 0, 1, 1 ]

> CT[2];
( 3, -1, 0, zeta(5)_5^3 + zeta(5)_5^2 + 1, -zeta(5)_5^3 -
  zeta(5)_5^2 )
> CT[2]:Minimal;
( 3, -1, 0, Z1, Z1#2 )
```

The character table is represented as a sequence of characters. Non-rational values in a character table are printed symbolically. This same printing can be forced on an individual character by using `Minimal` printing.

Example H91E2

The `CharacterTable` algorithm can handle quite large groups. We illustrate this by computing the character table of the almost simple group $P\Gamma U_5(4)$.

```
> G := PGammaU(5,4);
> G;
Permutation group G acting on a set of cardinality 17425
Order = 2^22 * 3^2 * 5^5 * 13 * 17 * 41
> time CT := CharacterTable(G);
Time: 205.150
> #CT;
160
> Degree(CT[160]);
5227500
```

91.2 Character Ring Operations

91.2.1 Related Structures

`Parent(R)` `Category(R)`

`Group(R)`

Given the ring R of class functions on a finite group G, return G.

`Centre(x)`

The centre of the character x of G, i.e. the subgroup of G consisting of those classes C of G for which $|x(g)|$, g in C, is equal to the degree of x.

`CoefficientField(x)`

The (minimal) coefficient field $\mathbf{Q}_m$ of the class function x.

`Kernel(x)`

The kernel of the character x of G, i.e. the normal subgroup of G consisting of those elements g for which $x(g) = x(1)$.

91.3 Element Operations

91.3.1 Arithmetic

In the list of arithmetic operations below x and y denote class functions in the same ring, and a denotes a scalar, which is any element coercible into a cyclotomic field. Also, j denotes an integer.

`+ y` `- y`

`x + y` `x - y` `x * y`

`a * x` `x ^ j`

91.3.2 Predicates and Booleans

The following Boolean-valued functions are available. Note that with the exception of `in`, `notin`, `IsReal` and `IsFaithful`, these functions use the table of irreducible characters, which will be created if it is not yet available.

`x in y`

Returns `true` if the inner product of class functions x and y is non-zero, otherwise `false`. If x is irreducible and y is a character, this tests whether or not x is a constituent of y.

`x notin y`

Returns `true` if the inner product of class functions x and y is zero, otherwise `false`. If x is irreducible and y is a character, this tests whether or not x is not a constituent of y. Returns `true` if the character x is not a constituent of the character y, otherwise `false`.

`a in F` `a notin F`

`x eq y` `x ne y`

`IsCharacter(x)`

Returns `true` if the class function x is a character, otherwise `false`. A class function is a character if and only if all inner products with the irreducible characters are non-negative integers.

`IsGeneralizedCharacter(x)`

Returns `true` if the class function x is a generalized character, otherwise `false`. A class function is a generalized character if and only if all inner products with the irreducible characters are integers.

`IsIrreducible(x)`

Returns `true` if the character x is an irreducible character, otherwise `false`.

`IsLinear(x)`

Returns `true` if the character x is a linear character, otherwise `false`.

`IsFaithful(x)`

Returns `true` if the character x is faithful, i.e. has trivial kernel, otherwise `false`.

`IsReal(x)`

Returns `true` if the character x is a real character, i.e. takes real values on all of the classes of G, otherwise `false`.

`IsOne(x)` `IsMinusOne(x)` `IsZero(x)`

91.3.3 Accessing Class Functions

In this subsection T is a character table, and x is any class function. A character table is an enumerated sequence of characters that has a special print function attached. In particular, its entries can be accessed with the ordinary sequence indexing operations.

`T[i]`

Given the table T of ordinary characters of G, return the i-th character of G, where i is an integer in the range $[1...k]$.

`T[i][j]`

The value of the i-th irreducible character (from the character table T) on the j-th conjugacy class of G.

`#T`

Given a character table T (or any sequence of characters), return the number of entries.

`x(g)`

`g @ x`

The value of the class function x on the element g of G.

`x[i]`

The value of the class function x on the i-th conjugacy class of G.

```
#x
```

Given a class function x on G return its length (which equals the number of conjugacy classes of the group G).

91.3.4 Conjugation of Class Functions

```
x ^ g
```

Given a class function x on a normal subgroup N of the group G, and an element g of G, construct the conjugate class function x^g of x which is defined as follows: $x^g(n) = x(g^{-1}ng)$, for all n in N.

```
x ^ H
```

Given a class function x on a normal subgroup N of the group G, and a subgroup H of G, construct the sequence of conjugates of x under the action of the subgroup H. The action of an element of H on x is that defined in the previous function.

```
GaloisConjugate(x, j)
```

Let $\mathbf{Q}(x)$ be the subfield of $\mathbf{Q}(\zeta_m)$ generated by $\mathbf{Q}$ and the values of the G-character x. This function returns the Galois conjugate x^j of x under the action of the element of the Galois group $\mathrm{Gal}(\mathbf{Q}(x)/\mathbf{Q})$ determined by the integer j. The integer j must be coprime to m.

```
GaloisOrbit(x)
```

Let $\mathbf{Q}(x)$ be the subfield of $\mathbf{Q}(\zeta_m)$ generated by $\mathbf{Q}$ and the values of the G-character x. This function returns the sequence of Galois conjugates of x under the action of the Galois group $\mathrm{Gal}(\mathbf{Q}(x)/\mathbf{Q})$.

```
ClassPowerCharacter(x, j)
```

Given a class function x on the group G and a positive integer j, construct the class function x^j which is defined as follows: $x^j(g) = x(g^j)$.

91.3.5 Functions Returning a Scalar

```
Degree(x)
```

The degree of the class function x, i.e. the value of x on the identity element of G.

```
InnerProduct(x, y)
```

The inner product of the class functions x and y, where x and y are class functions belonging to the same character ring.

```
Order(x)
```

Given a linear character of the group G, determine the order of x as an element of the group of linear characters of G.

```
Norm(x)
```

Norm of the class function x (which is the inner product with itself).

```
Schur(x, k)
Indicator(x)
```

Given class function x and a positive integer k, return the generalised Frobenius–Schur indicator which is defined as follows: Suppose g is some element of G, and set $T_k(g) = |\{h \in G | h^k = g\}|$. The value of `Schur(x, k)` is the coefficient a_x in the expression $T_k = \sum_{x \in \mathrm{Irr}(G)} a_x x$.

The call `Indicator(x)` is equivalent to `Schur(x,2)`.

```
StructureConstant(G, i, j, k)
```

The structure constant $a_{i,j,k}$ for the centre of the group algebra of the group G. If K_i is the formal sum of the elements of the i-th conjugacy class, $a_{i,j,k}$ is defined by the equation $K_i * K_j = \sum_k a_{i,j,k} * K_k$.

91.3.6 The Schur Index

Magma incorporates functions for computing the Schur index of an ordinary irreducible character over various number fields and local fields. The routines below are all based on the function `SchurIndices(x)`, which computes the Schur Indices of the given character over all the completions of the rationals.

The algorithm is based on calculations with characters, groups and fields, and does not compute representations.

The algorithm was devised by Gabi Nebe and Bill Unger, with code written by Bill Unger. The extension to compute a Schur index over a number field was written by Claus Fieker.

```
SchurIndex(x)
SchurIndex(x, F)
```

The Schur index of the character x over the given field. When no field is given, the Schur index over the rationals is returned. The character x must be a complex irreducible character. The field F must be an absolute number field.

```
SchurIndices(x)
SchurIndices(x, F)
SchurIndices(C, s, F)
```

Compute the Schur indices of the character x over the completions of the given field. The character x must be a complex irreducible character. The field F must be an absolute number field. When no field is specified the rational field is assumed. The last form takes the character field, C, and the output from `SchurIndices(x)`, s, as well as a number field. This is sufficient to compute the Schur indices over the number field without repeating group and character computations when a number of fields are being considered for one character.

The return value is a sequence of pairs. Each pair gives a completion at which the Schur index is not 1, followed by the Schur index over the complete field. For the rational field, a completion is specified by an integer. The integer zero specifies the archimedean completion (the real numbers), while a prime p specifies the p-adic field $\mathbf{Q}_p$. When a number field is given, the completions are specified by a place of the field, an object of type `PlcNumElt`.

If the character has Schur index 1 over the given field the return value will be an empty sequence. Otherwise the Schur index over the given field is the least common multiple of the second entries of the tuples returned.

Example H91E3

We first look at the faithful irreducible character of the Dihedral group of order 8. It has Schur index 1.

```
> T := CharacterTable(SmallGroup(8, 3));
> T[5];
( 2, -2, 0, 0, 0 )
> SchurIndex(T[5]);
1
> SchurIndices(T[5]);
[]
```

The corresponding character of the quaternion group of order 8 has non-trivial Schur index.

```
> T := CharacterTable(SmallGroup(8, 4));
> T[5];
( 2, -2, 0, 0, 0 )
> SchurIndex(T[5]);
2
> SchurIndices(T[5]);
[ <0, 2>, <2, 2> ]
```

The Schur index is 2 over the real numbers and Q_2. For all odd primes p, the Schur index over Q_p is 1. We look at the Schur index of this character over some number fields. First we look at some cyclotomic fields.

```
> [SchurIndex(T[5], CyclotomicField(n)):n in [3..20]];
[ 1, 1, 1, 1, 2, 1, 1, 1, 1, 1, 1, 2, 1, 1, 1, 1, 1, 1 ]
> SchurIndices(T[5], CyclotomicField(7));
[ <Place at Prime Ideal
Two element generators:
  [2, 0, 0, 0, 0, 0]
  [1, 1, 0, 1, 0, 0], 2>, <Place at Prime Ideal
Two element generators:
  [2, 0, 0, 0, 0, 0]
  [1, 0, 1, 1, 0, 0], 2> ]
```

The cyclotomic field of order 7 gives Schur index 2. An archimedean completion of this field is necessarily the field of complex numbers, hence no infinite places give Schur index greater than 1. There are now two 2-adic completions which give Schur index 2.

```
> P<t> := PolynomialRing(Rationals());
> F := ext<Rationals()|t^3-2>;
> SchurIndex(T[5], F);
2
> SchurIndices(T[5], F);
[ <1st place at infinity, 2>, <Place at Prime Ideal
Two element generators:
  [2, 0, 0]
  [0, 1, 0], 2> ]
```

For the non-normal field F, one archimedean completion is real, the other complex. Thus the real field features in the output of `SchurIndices`, along with the 2-adic completion.

Example H91E4

We will use a general construction for a character with given Schur index over the rationals to construct a character with Schur index 6. Given an integer $n \geq 1$, we select a prime p such that $p = kn + 1$ where k and n are coprime. We take an integer a such that a has order n modulo p. We then consider the metacyclic group

$$G = \langle < x, y | x^{n^2}, y^p, y^x = y^a \rangle.$$

The order of G is n^2p. The subgroup of G generated by x^n and y is cyclic, normal and self-centralizing in G with order np. If λ is any faithful linear character of this subgroup, then λ^G is an irreducible character of G with Schur index n over the rational field. The correctness of this construction is proved in Lemma 3 of [Tur01].
We construct G in two stages. First as a finitely presented group as described above. Then we convert to a PC-presentation for further computations. We take $n = 6$, $p = 7$ and $a = 3$.

```
> G1 := Group<x,y|x^36, y^7, y^x = y^3>;
> G, f := SolubleQuotient(G1, 36*7);
> x := f(G1.1); y := f(G1.2);
> C := sub<G|x^6,y>;
> IsCyclic(C);
true
> IsNormal(G, C);
true
> Centralizer(G,C) eq C;
true
> exists(l){l:l in LinearCharacters(C)|IsFaithful(l)};
true;
> c := Induction(l, G);
> IsIrreducible(c);
true
> Degree(c);
```

```
6
> CharacterField(c);
Cyclotomic Field of order 3 and degree 2 in sparse
representation
> SchurIndex(c);
6
```

The construction of the previous example is used in the following two intrinsics.

`SchurIndexGroup(n: parameters)`

`Prime` RngIntElt *Default :*

Return a group having a faithful character with Schur index n over the rational field. The construction used is as in the previous example. The parameter `Prime` may be used to supply the prime p. (The necessary conditions on p are not checked. If these conditions are not met, an error is possible.) If `Prime` is not set, then the least prime meeting the conditions is used.

`CharacterWithSchurIndex(n: parameters)`

`Prime` RngIntElt *Default :*

Return a character with Schur index n over the rational field. The construction used is as in the previous example. The second return value is the group of the character, equal to `SchurIndexGroup(n)`. The parameter `Prime` is as for `SchurIndexGroup`.

91.3.7 Attribute

`AssertAttribute(x, "IsCharacter", b)`

Procedure that, given a class function x and a Boolean value b, stores with x the information that the value of the predicate `IsCharacter(x)` equals b.

91.3.8 Induction, Restriction and Lifting

`Induction(x, G)`

Given a class function x on the subgroup H of the group G, construct the class function obtained by induction of x to G. Note that if x is a character of H, then `Induction(x, G)` will return a character of G.

The `Induction` command may also be used to induce a sequence of characters of a particular subgroup (such as a character table) to the given supergroup.

`LiftCharacter(c, f, G)`

Given a class function c of the quotient group Q of the group G and the natural homomorphism $f : G \to Q$, lift c to a class function of G.

```
LiftCharacters(T, f, G)
```

Given a sequence T of class functions of the quotient group Q of the group G and the natural homomorphism $f : G \to Q$, lift T to a sequence of corresponding class functions of G. Since a character table is just a sequence of class functions which is printed in a special way, this intrinsic may also be applied to it.

```
Restriction(x, H)
```

Given a class function x on the group G and a subgroup H of G, construct the restriction of x to H (a class function). Note that if x is a character of G, then `Restriction(x, H)` will return a character of H.

91.3.9 Symmetrization

See [Mur58] or [Fra82] for more details.

```
Symmetrization(x, p)
```

Given a class function x and a partition p of n ($2 \leq n \leq 6$), this function returns the symmetrized character with respect to p; the partition must be specified in the form of a sequence of positive integers (adding up to n).

```
OrthogonalComponent(x, p)
```

Given a class function x and a partition p of n ($2 \leq n \leq 6$), this function returns the Murnaghan component of the orthogonal symmetrization of x with respect to p; the partition must be specified in the form of a sequence of positive integers (adding up to n). Here x may not be a linear character, and its Frobenius–Schur indicator must be 1.

```
SymplecticComponent(x, p)
```

Given a class function x and a partition p of n ($2 \leq n \leq 6$), this function returns the Murnaghan component of the symplectic symmetrization of x with respect to p; the partition must be specified in the form of a sequence of positive integers (adding up to n). Here x may not be a linear character, and its Frobenius–Schur indicator must be -1.

```
SymmetricComponents(x, n)
```

Given a class function x and an integer n, return the set of symmetrizations of x by all partitions of m with $2 < m \leq n \leq 5$.

```
OrthogonalComponents(x, n)
```

Given a class function x, return the set of Murnaghan components for orthogonal symmetrizations of x by all partitions of m with $2 < m \leq n \leq 6$. Here x may not be a linear character, and its Frobenius–Schur indicator must be 1.

```
SymplecticComponents(x, n)
```

Given a class function x, return the set of Murnaghan components for symplectic symmetrizations of x by all partitions of m with $2 < m \leq n \leq 5$. Here x may not be a linear character, and its Frobenius–Schur indicator must be -1.

91.3.10 Permutation Character

```
PermutationCharacter(G)
```

Given group G represented as a permutation group, construct the character of G afforded by the defining permutation representation of G.

```
PermutationCharacter(G, H)
```

Given a group G and some subgroup H of G, construct the character of G afforded by the permutation representation of G given by the action of G on the right cosets of H in G.

91.3.11 Composition and Decomposition

```
Composition(T, q)
```

Given a sequence or table of characters T for the group G and a sequence q of k elements of $\mathbf{Q}(\zeta_m)$ (possibly $\mathbf{Q}$), create the class function $q_1 * T_1 + \cdots + q_k * T_k$, where T_i is the i-th character in T.

```
Decomposition(T, y)
```

Given a sequence or table of class functions T for G of length l and a class function y on G, attempt to express y as a linear combination of the elements of T.

The function returns two values: a sequence $q = [q_1, \ldots, q_l]$ of cyclotomic field elements and a class function z. For $1 \leq i \leq l$, the i-th term of q is defined to be the ratio of inner products $(y, T_i)/(T_i, T_i)$, where T_i is the i-th entry of T. The sequence q determines a class function $x = q_1 \cdot T_1 + \cdots + q_l \cdot T_l$ which will equal y if T is the complete table of irreducible characters. The difference $z = y - x$ is the second return value. If the entries in T are mutually orthogonal, then z is the zero class function if and only if y is a linear combination of the T_i.

91.3.12 Finding Irreducibles

A common approach to finding the irreducible characters of a group is to start with an irreducible character and generate new characters by applying `SymmetricComponents`, `OrthogonalComponents` or `SymplecticComponents`. Then, by examining norms and inner products, it is often possible to identify irreducible characters or at least characters with smaller norms. There are two Magma intrinsics available to help with this task.

```
RemoveIrreducibles(I, C)
```

Remove occurrences of the irreducible characters in the sequence I from the characters in the sequence C and look for characters of norm 1 among the reduced characters. Return a sequence of new irreducibles found and the sequence of reduced characters.

```
ReduceCharacters(I, C)
```

Make the norms of the characters in the sequence C smaller by computing the differences of appropriate pairs. Return a sequence of new irreducibles found and a sequence of reduced characters.

Example H91E5

We conclude this section with an example, showing how the above functions can be used to construct the character table for A_5 (compare Isaacs, p64), using only characters on subgroups.

```
> A := AlternatingGroup(GrpPerm, 5);
> R := CharacterRing(A);
```

The first character will be the principal character

```
> T1 := R ! 1;
> T1;
( 1, 1, 1, 1, 1 )
```

Next construct the permutation character

```
> pc := PermutationCharacter(A);
> T2 := pc - T1;
> InnerProduct(pc, T1), InnerProduct(T2, T2);
1 1
> T2;
( 4, 0, 1, -1, -1 )
```

It follows that pc - T1 is an irreducible character

```
> B := Stabilizer(A, 5);
> r := RootOfUnity(3, CyclotomicField(3));
> S := CharacterRing(B);
> lambda := S ! [1, 1, r, r^2 ];
> IsLinear(lambda);
true
```

This defines a linear character on a subgroup of index 5 in A

```
> T3 := Induction(lambda, A);
> InnerProduct(T3, T3);
1
> T3;
```

```
( 5, 1, -1, 0, 0 )
```

Finally we use characters on the cyclic subgroup of order 5:

```
> K := sub<A |  (1,2,3,4,5) >;
> Y := CharacterTable(K);
> Y;
```

```
Character Table of Group K
--------------------------

-------------------------------
Class |   1    2    3    4    5
Size  |   1    1    1    1    1
Order |   1    5    5    5    5
-------------------------------
p  =  5   1    1    1    1    1
-------------------------------
X.1   +   1    1    1    1    1
X.2   0   1   Z1 Z1#2 Z1#3 Z1#4
X.3   0   1 Z1#2 Z1#4   Z1 Z1#3
X.4   0   1 Z1#3   Z1 Z1#4 Z1#2
X.5   0   1 Z1#4 Z1#3 Z1#2   Z1

Explanation of Symbols:
-----------------------

# denotes algebraic conjugation, that is,
# k indicates replacing the root of unity w by w^k

Z1     = -1 - zeta_5 - zeta_5^2 - zeta_5^3
```

```
> mu := Induction(Y[2], A);
```

We subtract what we already know from mu and get a new irreducible. We use decomposition with respect to a sequence.

```
> _, T4 := Decomposition([T1, T2, T3], mu);
> InnerProduct(T4, T4);
1
> T4;
( 3, -1, 0, (1 + zeta_5^2 + zeta_5^3), (-zeta_5^2 - zeta_5^3) )
> T5 := GaloisConjugate(T4, 2);
> T5;
( 3, -1, 0, (-zeta_5^2 - zeta_5^3), (1 + zeta_5^2 + zeta_5^3) )
```

Compare this to the standard character table:

```
> CharacterTable(A);
```

```
Character Table of Group A
--------------------------

---------------------------
```

```
Class |   1  2  3     4     5
Size  |   1 15 20    12    12
Order |   1  2  3     5     5
---------------------------
p  =  2   1  1  3     5     4
p  =  3   1  2  1     5     4
p  =  5   1  2  3     1     1
---------------------------
X.1   +   1  1  1     1     1
X.2   +   3 -1  0    Z1  Z1#2
X.3   +   3 -1  0  Z1#2    Z1
X.4   +   4  0  1    -1    -1
X.5   +   5  1 -1     0     0

Explanation of Symbols:
-----------------------

#  denotes algebraic conjugation, that is,
# k indicates replacing the root of unity w by w^k

Z1     =(1 + zeta_5^2 + zeta_5^3)
```

91.3.13 Brauer Characters

Magma has some support for the calculation of Brauer characters. These functions are noted in this section. We anticipate considerable change to the functionality described here in the near future.

A Brauer character modulo p in Magma is represented as a class function (that is, element of a character ring) which is zero on p-singular group elements. In this format the standard character operations of addition, multiplication, induction and restriction all apply directly to Brauer characters as they do to other class functions.

Note that problems associated with choice of lifting from finite fields to complex roots of unity have not yet been dealt with.

```
BrauerCharacter(x, p)
```

The Brauer character modulo the prime p obtained by setting the value of x on p-singular elements to be zero.

```
Blocks(T, p)
```

When T is the full ordinary character table of a group, return the partition of T into p-blocks, where p is a given prime. The partition is returned as a sequence of sets of integers which give the blocks by the positions of the characters in T. The second return value is the corresponding sequence of defects of the blocks. The blocks are ordered first by decreasing defect, second by first character in the block.

Example H91E6

We give an example of the use of these Brauer character functions. We consider the 3-modular characters of the Higman-Sims simple group.

```
> load hs176;
> T := CharacterTable(G);
> Blocks(T,3);
[
  { 1, 2, 5, 10, 18, 19, 21, 23, 24 },
  { 3, 4, 6, 7, 11, 12, 14, 15, 20 },
  { 8, 13, 16 },
  { 9 },
  { 17 },
  { 22 }
]
[ 2, 2, 1, 0, 0, 0 ]
```

The characters $T[8], T[13], T[16]$ are the ordinary irreducible characters in a 3-block of defect one. In such a small block the two ordinary irreducibles of minimal degree will restrict to modular irreducibles.

```
> [Degree(T[i]): i in [8, 13, 16]];
[ 231, 825, 1056 ]
> BrauerCharacter(T[8], 3);
( 231, 7, -9, 0, 15, -1, -1, 6, 1, 1, 0, 0, 0, -1, -1, -1,
2, 1, 0, 0, 0, 0, 0, 0 )
> BrauerCharacter(T[13], 3);
( 825, 25, 9, 0, -15, 1, 1, 0, -5, 0, 0, 0, -1, 1, 1, 1, 0,
-1, 0, 0, 0, 0, 0, 0 )
> $1 + $2 eq BrauerCharacter(T[16], 3);
true
```

The projective indecomposable characters corresponding to these Brauer irreducible characters are as follows.

```
> T[8] + T[16];
( 1287, 39, -9, 0, 15, -1, -1, 12, -3, 2, 0, 0, -1, -1, -1,
-1, 4, 1, 0, 0, 0, 0, 0, 0 )
> T[13] + T[16];
( 1881, 57, 9, 0, -15, 1, 1, 6, -9, 1, 0, 0, -2, 1, 1, 1, 2,
-1, 0, 0, 0, 0, 0, 0 )
```

91.4 Bibliography

[**Con90**] S. B. Conlon. Calculating characters of p-groups. *J. Symbolic Comp.*, 9:535–550, 1990.

[**Dix67**] J. D. Dixon. High–speed computation of group characters. *Numerische Mathematik*, 10:446–450, 1967.

[**Fra82**] J. S. Frame. Recursive computation of tensor power components. *Bayreuth. Math. Schr.*, (10):153–159, 1982.

[**Mur58**] F. D. Murnaghan. The orthogonal and symplectic groups. *Comm. Dublin Inst. Adv. Studies. Ser. A, no.*, 13:146, 1958.

[**Sch90**] G. J. A. Schneider. Dixon's Character Table Algorithm Revisited. *J. Symbolic Computation*, 9:601–606, 1990.

[**Tur01**] A. Turull. Schur indices of perfect groups. *Proc. Amer. Math. Soc.*, 130(2): 367–370, 2001.

[**Ung06**] W.R. Unger. Computing the character table of a finite group. *J. Symbolic Comp.*, 41(8):847–862, 2006.

92 REPRESENTATIONS OF SYMMETRIC GROUPS

Chapter 92
REPRESENTATIONS OF SYMMETRIC GROUPS

92.1 Introduction

This chapter describes functions available in Magma for computations concerning the non modular representation theory of the symmetric group and the alternating group. It is possible to compute different matrix representations, complete character tables using special routines, single irreducible characters or the value of an irreducible character on an element of the group.

92.2 Representations of the Symmetric Group

For the symmetric group of degree n the irreducible representations can be indexed by partitions of weight n. For more information on partitions see Section 145.2.

92.2.1 Integral Representations

It is possible to define representing matrices of the symmetric group over the integers.

```
SymmetricRepresentation(pa, pe)
```

`Al` MonStgElt *Default : "JamesKerber"*

Given a partition *pa* of weight n and a permutation *pe* in a symmetric group of degree n, return an irreducible representing matrix for *pe*, indexed by *pa*, over the integers. If `Al` is set to the default `"JamesKerber"` then the method described in [JK81] is used. If `Al` is set to `"Boerner"` the method described in the book of Boerner [Boe67] is used. If `Al` is set to `"Specht"` then the method used is a direct implementation of that used by Specht in his paper from 1935 [Spe35].

Example H92E1

We compute a representing matrix of a permutation using two different algorithms and check whether the results have the same character.

```
> a:=SymmetricRepresentation([3,2],Sym(5)!(3,4,5) : Al := "Boerner");a;
[ 0  0  1 -1  0]
[ 1  0  0 -1  0]
[ 0  1  0 -1  0]
[ 0  0  0 -1  1]
[ 0  0  0 -1  0]
> b:=SymmetricRepresentation([3,2],Sym(5)!(3,4,5) : Al := "Specht");b;
```

```
[ 0  1  0 -1  0]
[ 0  0  1  0 -1]
[ 1  0  0  0  0]
[ 0  0  0  0 -1]
[ 0  0  0  1 -1]
> IsSimilar(Matrix(Rationals(), a), Matrix(Rationals(), b));
true
```

The matrices are similar as they should be.

92.2.2 The Seminormal and Orthogonal Representations

The seminormal and orthogonal representations involve matrices which are not necessarily integral. The method Magma uses to construct these matrices is described in [JK81, Section 3.3];

```
SymmetricRepresentationSeminormal(pa, pe)
```

Given a partition *pa* of weight n and a permutation *pe* in a symmetric group of degree n, return the matrix of the seminormal representation for *pe*, indexed by *pa*, over the rationals.

```
SymmetricRepresentationOrthogonal(pa, pe)
```

Given a partition *pa* of weight n and a permutation *pe* in a symmetric group of degree n, return the matrix of the orthogonal representation for *pe*, indexed by *pa*. An orthogonal basis is used to compute the matrix which may have entries in a cyclotomic field.

Example H92E2

We compare the seminormal and orthogonal representations of a permutation and note that they are similar.

```
> g:=Sym(5)!(3,4,5);
> a:=SymmetricRepresentationSeminormal([3,2],g);a;
[-1/2    0 -3/4    0    0]
[   0  1/2    0  3/4    0]
[   1    0 -1/2    0    0]
[   0  1/3    0 -1/6  8/9]
[   0    1    0 -1/2 -1/3]
> b:=SymmetricRepresentationOrthogonal([3,2],g);b;
[-1/2 0 zeta(24)_8^2*zeta(24)_3 + 1/2*zeta(24)_8^2 0 0]
[0 1/2 0 -zeta(24)_8^2*zeta(24)_3 - 1/2*zeta(24)_8^2 0]
[-zeta(24)_8^2*zeta(24)_3 - 1/2*zeta(24)_8^2 0 -1/2 0 0]
[0 -1/3*zeta(24)_8^2*zeta(24)_3 - 1/6*zeta(24)_8^2 0
 -1/6 2/3*zeta(24)_8^3 - 2/3*zeta(24)_8]
[0 2/3*zeta(24)_8^3*zeta(24)_3 + 1/3*zeta(24)_8^3
+ 2/3*zeta(24)_8*zeta(24)_3 + 1/3*zeta(24)_8 0
```

```
    -1/3*zeta(24)_8^3 + 1/3*zeta(24)_8 -1/3]
> IsSimilar(a,b);
true
```

They should both be of finite order, 3.

```
> IsOne(a^Order(g));
true
> IsOne(b^Order(g));
true
>
```

92.3 Characters of the Symmetric Group

92.3.1 Single Values

The method used to compute the value of a character on a permutation is the recursion formula of Murnaghan and Nakayama [JK81, p. 60], except when computing the value of a character on the identity permutation a formula for the dimension of the representation indexed by a partition is used, [JK81, p. 56].

```
SymmetricCharacterValue(pa, pe)
```

Computes the value of the irreducible character of the symmetric group of degree n indexed by the partition *pa* of weight n on the permutation *pe*. When computing the value of the character on the identity permutation, i.e. the degree of the character, the dimension (hook) formula is used on the partition *pa*.

92.3.2 Irreducible Characters

```
SymmetricCharacter(pa)
```

Return the character of the representation of the symmetric group of degree n indexed by the partition *pa* where *pa* has weight n.

92.3.3 Character Table

```
SymmetricCharacterTable(d)
```

Return the character table of the symmetric group of degree d.

92.4 Representations of the Alternating Group

92.5 Characters of the Alternating Group

There is a special routine which computes the character table of the alternating group, as well as routines which compute values of alternating characters and the characters themselves. These routines make use of the fact that in most cases the irreducible characters of the symmetric group, which can be computed quickly, are also irreducible in the alternating group. So the irreducible characters of the alternating group may be indexed by partitions in the same way as those of the symmetric group. As the restriction of the irreducible character indexed by the partition λ is equal to the restriction of the character indexed by the conjugate partition, we only need to take one from each of these pairs of partitions to form a full set of irreducible characters. When a partition is conjugate to itself the character of the symmetric group indexed by that partition is no longer irreducible but is the sum of two irreducibles. This method is described in [JK81].

92.5.1 Single Values

```
AlternatingCharacterValue(pa, pe)
```

Return the value of the character of the alternating group of degree n indexed by the partition *pa* of weight n on the permutation *pe*. The partition *pa* and its conjugate should be distinct.

```
AlternatingCharacterValue(pa, i, pe)
```

Return the value of the ith character of the alternating group of degree n indexed by the self conjugate partition *pa* of weight n on the permutation *pe*. Since there are two possible irreducible characters indexed by such partitions i must be either 1 or 2.

92.5.2 Irreducible Characters

```
AlternatingCharacter(pa)
```

Return the character of the alternating group of degree n indexed by the partition *pa* of weight n. The partition *pa* and its conjugate should be distinct.

```
AlternatingCharacter(pa, i)
```

Return the ith character of the alternating group of degree n indexed by the self conjugate partition *pa* of weight n. Since there are two possible irreducible characters indexed by such partitions i must be either 1 or 2.

92.5.3 Character Table

```
AlternatingCharacterTable(d)
```

Returns the character table of the alternating group of degree d.

92.6 Bibliography

[**Boe67**] H. Boerner. *Darstellungen von Gruppen. 2. Aufl.* Berlin-Heidelberg-New York: Springer-Verlag. XIV, 317 S. , 1967.

[**JK81**] Gordon James and Adalbert Kerber. *The representation theory of the symmetric group.* Addison-Wesley Publishing Co., Reading, Mass., 1981. With a foreword by P. M. Cohn, With an introduction by Gilbert de B. Robinson.

[**Spe35**] Wilhelm Specht. Die irreduziblen Darstellungen der symmetrischen Gruppe. *Math. Z.*, 39:696–711, 1935.

93 MOD P GALOIS REPRESENTATIONS

Chapter 93
MOD P GALOIS REPRESENTATIONS

93.1 Introduction

This package provides tools to work with φ-modules over $k((u))$ where k is a finite field, and representations of the absolute Galois group of $k((u))$ with coefficients in a finite field. The main functionality of the package computes the semisimplification of a given φ-module, and the semisimplification of the Galois representation that is naturally attached to it. In particular, the slopes of the φ-module, corresponding to the tame inertia weights of the Galois representation, can be computed using this package.

93.1.1 Motivation

Let K be a p-adic field and let G_K be the absolute Galois group of K. Representations of this group naturally arise from geometry, namely from the p-adic étale cohomology of a scheme over K.

The study of these representations is a central topic in arithmetic, and a motivation for creating this package is the following: let V be a $\mathbf{Q}_p$-representation of G_K, *i.e.* a $\mathbf{Q}_p$-vector space endowed with a continuous, linear action of G_K. Now let $T \subset V$ be any $\mathbf{Z}_p$-lattice stable under the action of G_K. There always exists such a lattice. Moreover, the quotient T/pT has a natural structure of $\mathbf{F}_p$-representation of G_K. This representation depends on the choice of T, but its semisimplification $(T/pT)^{ss}$ does not, according to the Brauer-Nesbitt theorem. Recall the semisimplification of a representation is the direct sum of the composition factors appearing in any Jordan-Holder sequence of this representation. Therefore, it is an interesting question to determine properties of $(T/pT)^{ss}$ in terms of V. Although the Fontaine-Laffaille theory completely addresses this question for some V, the general case remains an open question. Some computations concerning this problem can be performed in Magma using this package.

93.1.2 Definitions

Let k be a finite field of characteristic p, and let $K = k((u))$ be the field of Laurent series with coefficients in k. Let $s \geq 0$ and $b \geq 2$ be integers. We define a “Frobenius” map σ on K by the following formula:

$$\sigma\left(\sum_{i\in\mathbf{Z}} a_i u^i\right) = \sum_{i\in\mathbf{Z}} a_i^{p^s} u^{bi}.$$

A φ-module over K is the data of a finite-dimensional K-vector space D, endowed with an endomorphism $\varphi : D \to D$ that is semilinear with respect to σ. This means that for all $\lambda \in K$, $x \in D$, we have the identity $\varphi(\lambda x) = \sigma(\lambda)\varphi(x)$.

A φ-module is said to be étale if the map φ is injective. A φ-module can be described by the matrix representing the action of φ on some basis of D, and it is étale if and only if this matrix is invertible.

93.1.3 Classification of φ-modules

Some φ-modules play a crucial role in the theory because they are the simple objects in the category of étale φ-modules over the maximal unramified extension K^{ur} of K.

Let $d \geq 1$, $h \in \mathbf{Z}$, $\lambda \in \bar{k}$. We define the φ-module $D(d, s, \lambda)$ as the φ-module of dimension d whose matrix in some basis is the companion matrix of the polynomial $T^d - u^h$. We also write $D(d, h) = D(d, h, 1)$. Note that in general there are several ways to extend the action of σ on K^{ur}, but we may only distinguish the cases where σ acts as identity on k, and the case where is does not. We say that a couple (d, h) is reduced if there is no divisor d' of d (except d) such that $\frac{b^{d'}-1}{b^d-1}$ is a divisor of h. The main classification results are the following:

If $\sigma \neq id$, the simple objects of the category of étale φ-modules over K^{ur} are the $D(d, h)$ for (d, h) reduced, and if $\sigma = id$, the simple objects of the category of étale φ-modules over K^{ur} are the $D(d, h, \lambda)$ for (d, h) reduced.

By definition, the slope of a simple φ-module isomorphic to $D(d, h, \lambda)$ is the rational number $\frac{h}{b^d-1}$, up to the equivalence relation "$x \sim y \Leftrightarrow \exists$m,n$\in$ N such that $b^m x - b^n y \in \mathbf{Z}$". With this equivalence relation, the definition does not depend on the choice of (d, h).

If D is a φ-module over K, the slopes of D are the collection of the slopes of the composition factors of $K^{ur} \otimes_K D$ (this notion does not depend on how σ is extended to K^{ur}). Note that even though the algorithms that we present can give decompositions over K, for most practical uses the knowledge of the slopes should be sufficient.

93.1.4 Connection with Galois Representations

Let us explain the link between Galois representations and φ-modules over K. In this section, we assume that σ is the classical Frobenius $x \mapsto x^p$. Let K^{sep} be a separable closure of K and let $G_K = Gal(K^{sep}/K)$ be the absolute Galois group of K.

A theorem of Katz states that there is an equivalence of categories between the étale φ-modules over K and the $\mathbf{F}_p$-representations of G_K.

Under this equivalence of categories, the φ-module $D(d, h)$ corresponds to the "fundamental character of level d" to the power h, ω_d^h, seen as a $\mathbf{F}_p$-representation. The figures of h in base p are called the tame inertia weights of the representation, because they describe the action of the tame inertia group on the representation. These weights can be recovered from the slope of the φ-module. It is worth noting that if F is a p-adic field whose residue field is k, and F_∞ is the extension of F generated by a compatible sequence of p^n-th roots of the uniformizer for all n, then G_{F_∞} is isomorphic to G_K. Moreover, the tame inertia weights of a $\mathbf{F}_p$-representation of G_F are the same as the tame inertia weights of its restriction to G_{F_∞}, seen as a representation of G_K. Hence, working with φ-modules will enable us to study representations of p-adic Galois groups.

93.2 φ-modules and Galois Representations in Magma

Let us now give an overview of the functionalities of the package.

93.2.1 φ-modules

93.2.1.1 Category

In Magma, φ-modules have type `PhiMod`. Elements of φ-modules have type `PhiModElt`.

93.2.1.2 Creation functions

```
PhiModule(M)
```

F SeqEnum *Default :* $[1, p]$

Create the φ-module whose matrix is given by M in some basis. The optional argument F describes the action of the Frobenius on coefficients: if $F = [s, b]$ then φ acts by $a \mapsto a^{p^s}$ on the residue field and maps the variable u to u^b. The default value is $[1, p]$ where p is the characteristic of the base field, corresponding to the absolute Frobenius.

```
ElementaryPhiModule(S,d,h)
```

F SeqEnum *Default :* $[1, p]$

Create the φ-module $D(d, s)$ whose matrix is the companion matrix of $T^d - u^s$.

```
PhiModuleElement(x,D)
```

Create the element of the φ-module D whose coordinates are given by the vector x.

93.2.1.3 Attributes of φ-modules

```
Dimension(D)
```

The dimension of a φ-module.

```
CoefficientRing(D)
```

The coefficient ring of a φ-module.

```
FrobeniusMatrix(D)
```

Return the matrix of the action of φ on D in the current basis.

93.2.1.4 Basic operations and properties of φ-modules

```
IsEtale(D)
```

Return `true` if the action of φ on D is injective. This is only possible up to the precision of the coefficient ring of D.

```
ChangePrecision(~D, prec)
```

Change the precision of the coefficient ring of D to `prec`.

```
DirectSum(D1, D2)
```

The direct sum of two φ-modules. The coefficient rings and Frobenius action on the coefficients must be the same.

```
BaseChange(~D, P)
```

Change the basis of D. The base change matrix is P, meaning that if G is the current matrix of φ, the new matrix will be $P^{-1}G\varphi(P)$.

```
RandomBaseChange(~D)
```

Randomly change the basis of D.

```
Phi(D, x)
```

Compute the image of $x \in D$ under the action of φ.

93.2.1.5 Reduction of φ-modules and Galois Representations

```
SemisimpleDecomposition(D)
```

Compute a Jordan-Holder sequence for the φ-module D. The result G, P, sl, pol is as follows: G is the matrix of φ in a basis where it is block upper triangular, with diagonal blocks corresponding to simple φ-modules. The matrix P gives the corresponding basis. The list *sl* is the list of the slopes of D, and the list *pol* is a list of polynomials. The isomorphism class of a simple block of G is determined by the corresponding slope and polynomial.

```
Slopes(D)
```

Compute the list of slopes of D (with multiplicities).

```
SSGaloisRepresentation(D)
```

Compute the semisimplification of the Galois representation corresponding to D.

93.2.2 Semisimple Galois Representations

This part is dedicated to the study of representations of absolute Galois groups of fields of the form $k((u))$ with k finite, and with coefficients in finite fields. The implementation is for semisimple representations, and these are described by their tame inertia weights and polynomials giving the action of the Frobenius on the unramified part.

93.2.2.1 Category

In Magma, semisimple Galois representations have type `SSGalRep`.

93.2.2.2 Creation functions

```
SSGaloisRepresentation(E,K,w,P)
```

Create the semisimple representation of the absolute Galois group of K with coefficients in E, tame inertia weights given by w, and action of the Frobenius described by the elements of the list P.

93.2.2.3 Basic operations

`CoefficientRing(V)`

The coefficient ring.

`FixedField(V)`

The fixed field of the absolute Galois group of which V is a representation.

`Weights(V)`

The tame inertia weights of V.

93.2.2.4 Representation associated to a φ-module

`SSGaloisRepresentation(D)`

If D is a φ-module over a field K of Laurent series this returns the semisimplification of the representation associated to D.

93.3 Examples

Example H93E1

We create two φ-modules D_1, D_2, build their direct sum, and compute its slopes and corresponding representation.

```
> k<e9> := GF(3,2);
> S<u> := LaurentSeriesRing(k,20);
> D1 := ElementaryPhiModule(S,3,2);
> D1;
Phi-module of dimension 3 over Laurent series field in u over GF(3^2)
with fixed relative precision 20 with matrix
[ O(u^20)         O(u^20)        u^2 + O(u^20)]
[ 1 + O(u^20)     O(u^20)             O(u^20) ]
[ O(u^20)        1 + O(u^20)          O(u^20) ] and Frobenius [1,3]
> M := Matrix(S,2,2,[0,k.1*u,1,0]);
> D2 := PhiModule(M);
> D2;
Phi-module of dimension 2 over Laurent series field in u over GF(3^2)
with fixed relative precision 20 with matrix
[ O(u^20)       e9*u^2 + O(u^20)]
[ 1 + O(u^20)            O(u^20) ] and Frobenius [1,3]
> D := DirectSum(D1,D2);
> Slopes(D);
[
  [2, 1],
  [3, 2]
]
> SSGaloisRepresentation(D);
```

```
Semisimple representation of the absolute Galois group of
Laurent series field in u over GF(3^2) with fixed relative
precision 20 with coefficients in Finite field of size 3 and
components [
[3, 18],
[2, 3]
]
```