The MAGMA package HeckeAlgebra

Gabor Wiese*

9th October 2007

Abstract

This is a short manual for the MAGMA package `HeckeAlgebra`. The author would like to thank Lloyd Kilford for very helpful suggestions.

1 Example

The following example explains the main functions of the package. Let us suppose that the file `HeckeAlgebra.mg` is stored in the current path. We first attach the package.

> Attach("HeckeAlgebra.mg");

We want the package to be silent, so we put:

> SetVerbose ("HeckeAlgebra",false);

If we would like more information on the computations being performed, we should have put the value ***true***. Since we want to store the data to be computed in a file, we now create the file.

> my_file := "datafile";

> CreateStorageFile(my_file);

Next, we would like to compute the Hecke algebras of the dihedral eigenforms of level 2039 over extensions of $\mathbb{F}_2$. First, we create a list of such forms.

> dih := DihedralForms(2039 : ListOfPrimes := [2], completely_split := false);

Now, we compute the corresponding Hecke algebras, print part of the computed data in a human readable format, and finally save the data to our file.

> for f in dih do
for> ha := HeckeAlgebras(f);
for> HeckeAlgebraPrint1(ha);
for> StoreData(my_file, ha);
for> end for;
Level 2039
Weight 2
Characteristic 2
Gorenstein defect 0
Dimension 1

*Institut für Experimentelle Mathematik, Universität Duisburg-Essen, Ellernstraße 29, 45326 Essen, Germany, e-mail: gabor.wiese@uni-due.de, http://maths.pratum.net

Number of operators used 3
Primes lt Hecke bound 68
Residue degree 2

Level 2039
Weight 2
Characteristic 2
Gorenstein defect 2
Dimension 6
Number of operators used 4
Primes lt Hecke bound 68
Residue degree 2

Level 2039
Weight 2
Characteristic 2
Gorenstein defect 0
Dimension 1
Number of operators used 3
Primes lt Hecke bound 68
Residue degree 6

Level 2039
Weight 2
Characteristic 2
Gorenstein defect 0
Dimension 1
Number of operators used 3
Primes lt Hecke bound 68
Residue degree 4

Level 2039
Weight 2
Characteristic 2
Gorenstein defect 0
Dimension 1
Number of operators used 3
Primes lt Hecke bound 68
Residue degree 4

Level 2039
Weight 2
Characteristic 2
Gorenstein defect 0

Dimension 1
Number of operators used 3
Primes lt Hecke bound 68
Residue degree 12

Level 2039
Weight 2
Characteristic 2
Gorenstein defect 0
Dimension 1
Number of operators used 3
Primes lt Hecke bound 68
Residue degree 12

With the function ***DihedralForms*** one may also compute exclusively representations that are completely split in the characteristic. The default is ***completely_split := true***. By the option ***bound*** we indicate primes up to which bound should be used as the characteristic. The following example illustrates this.

> dih1 := DihedralForms (431 : bound := 20);
> for f in dih1 do
for> ha := HeckeAlgebras(f);
for> HeckeAlgebraPrint1(ha);
for> StoreData(my_file, ha);
for> end for;
Level 431
Weight 2
Characteristic 2
Gorenstein defect 2
Dimension 4
Number of operators used 6
Primes lt Hecke bound 20
Residue degree 1

Level 431
Weight 11
Characteristic 11
Gorenstein defect 2
Dimension 4
Number of operators used 5
Primes lt Hecke bound 77
Residue degree 3

One can also compute icosahedral modular forms over extensions of $\mathbb{F}_2$, starting from an integer polynomial with Galois group A_5, as follows.

> R<x> := PolynomialRing(Integers());
> pol := x^5-x^4-780*x^3-1795*x^2+3106*x+344;
> f := A5Form(pol);
With this kind of icosahedral examples one has to pay attention to the conductor, as it can be huge. This polynomial has prime conductor. But conductors need not be square-free, in general.
> print Modulus(f'Character);
1951
So it's reasonable. We do the computation.
> ha := HeckeAlgebras(f);
> HeckeAlgebraPrint1(ha);
Level 1951
Weight 2
Characteristic 2
Gorenstein defect 0
Dimension 3
Number of operators used 3
Primes lt Hecke bound 66
Residue degree 4

Level 1951
Weight 2
Characteristic 2
Gorenstein defect 0
Dimension 6
Number of operators used 3
Primes lt Hecke bound 66
Residue degree 4

There are two forms, which is okay, since they come from a weight one form in two different ways and this case is not exceptional. We now save them, as always.
> StoreData(my_file, ha);
It is also possible to compute all forms at a given character and weight.
> eps := DirichletGroup(229,GF(2)).1;
> ha := HeckeAlgebras(eps,2);
> HeckeAlgebraPrint1(ha);
Level 229
Weight 2
Characteristic 2
Gorenstein defect 0
Dimension 1
Number of operators used 12
Primes lt Hecke bound 12
Residue degree 1

Level 229
Weight 2
Characteristic 2
Gorenstein defect 0
Dimension 2
Number of operators used 12
Primes lt Hecke bound 12
Residue degree 2

Level 229
Weight 2
Characteristic 2
Gorenstein defect 0
Dimension 4
Number of operators used 12
Primes lt Hecke bound 12
Residue degree 1

Level 229
Weight 2
Characteristic 2
Gorenstein defect 0
Dimension 2
Number of operators used 12
Primes lt Hecke bound 12
Residue degree 5

> StoreData(my_file,ha);

Next, we illustrate how one reloads what has been saved. One would like to type: ***load my_file;*** but that does not work. One has to do it as follows.

> load "datafile";

> mf := RecoverData(LoadIn,LoadInRel);

Now, ***mf*** contains a list of all algebra data computed before. There's a rather concise printing function, displaying part of the information, namely ***HeckeAlgebraPrint(mf);***.

One can also create a LaTeX longtable. The entries can be chosen in quite a flexible way. The standard usage is the following.

> HeckeAlgebraLaTeX(mf,"table.tex");

A short LaTeX file displaying the table is the following:

```
\documentclass[11pt]{article}
\usepackage{longtable}
\begin{document}
\input{table}
\end{document}
```

The table we created is this one:

Level	Wt	ResD	Dim	EmbDim	NilO	GorDef	#Ops	#(p<HB)	Gp
2039	2	2	1	0	0	0	3	68	D_3
2039	2	2	6	3	2	2	4	68	D_5
2039	2	6	1	0	0	0	3	68	D_9
2039	2	4	1	0	0	0	3	68	D_{15}
2039	2	4	1	0	0	0	3	68	D_{15}
2039	2	12	1	0	0	0	3	68	D_{45}
2039	2	12	1	0	0	0	3	68	D_{45}
431	2	1	4	3	1	2	6	20	D_3
431	11	3	4	3	1	2	5	77	D_7
1951	2	4	3	1	2	0	3	66	A_5
1951	2	4	6	2	3	0	3	66	A_5
229	2	1	1	0	0	0	12	12	
229	2	2	2	1	1	0	12	12	
229	2	1	4	1	3	0	12	12	
229	2	5	2	1	1	0	12	12	

In the examples of level 229 the image of the Galois representation as an abstract group is not know. That is due to the fact that we created these examples without specifying the Galois representation in advance.

It is possible to compute arbitrary Hecke operators on the local Hecke factors generated by ***HeckeAlgebras(·)***, as the following example illustrates.

> A,B,M,C := HeckeAlgebras(DirichletGroup(253,GF(2)).1,2 : over_residue_field := true);

Suppose that we want to know the Hecke operator T_{17} on the 4th local factor.

> i := 4;

> T := BaseChange(HeckeOperator(M,17),C[i]);

The coefficients are the eigenvalues (only one):

> Eigenvalues(T);

{ < $. 1^5, 8 > }

Let us remember the eigenvalue.

> e := SetToSequence(Eigenvalues(T))[1][1];

In order to illustrate the option ***over_residue_field***, we also compute the following:

> A1,B1,M1,C1 := HeckeAlgebras(DirichletGroup(253,GF(2)).1,2 : over_residue_field := false);

> T1 := BaseChange(HeckeOperator(M1,17),C1[i]);

> Eigenvalues(T1);

{}

The base field is strictly smaller than the residue field in this example and the operator ***T1*** cannot be diagonalised over the base field. We check that ***e*** is nevertheless a zero of the minimal polynomial of ***T1***.

> Evaluate(MinimalPolynomial(T1),e);

0

The precise usage of the package is described in the following sections.

2 Hecke algebra computation

2.1 The modular form format

In the package, modular forms are often represented by the following record.
ModularFormFormat := recformat <

Character	***: GrpDrchElt,***
Weight	***: RngIntElt,***
CoefficientFunction	***: Map,***
ImageName	***: MonStgElt,***
Polynomial	***: RngUPolElt***

>;

The fields ***Character*** and ***Weight*** have the obvious meaning. Sometimes, the image of the associated Galois representation is known as an abstract group. Then that name is recorded in ***ImageName***, e.g. ***A_5*** or ***D_3***. In some cases, a polynomial is known whose splitting field is the number field cut out by the Galois representation. Then the polynomial is stored in ***Polynomial***. The cases in which polynomials are known are usually icosahedral ones. The ***CoefficientFunction*** is a function from the integers to a polynomial ring. For all primes l different from the characteristic and not dividing the level of the modular form (i.e. the modulus of the ***Character***), the coefficient function should return the minimal polynomial of the l-th coefficient in the q-expansion of the modular form in question.

2.2 Dihedral modular forms

Eigenforms whose associated Galois representations take dihedral groups as images provide an important source of examples, in many contexts. These eigenforms are called *dihedral*. The big advantage is that their Galois representation, and hence their q-coefficients, can be computed using class field theory. That enables one to exhibit Galois representations in the context of modular forms with certain number theoretic properties. The property for which these functions were initially created is that the representations should be unramified in the characteristic, say p, and that p is completely split in the number field cut out by the representation.

We consider dihedral representations whose determinant is the Legendre symbol of a quadratic field $\mathbb{Q}(\sqrt{N})$. The representations produced by the functions to be described are obtained by induction of an unramified character χ of $\mathbb{Q}(\sqrt{N})$ whose conjugate by the non-trivial element of the Galois group of $\mathbb{Q}(\sqrt{N})$ over $\mathbb{Q}$ is assumed to be χ^{-1}.

intrinsic GetLegendre (N :: RngIntElt, K :: FldFin) -> GrpDrchElt
For an odd positive integer ***N***, this function returns the element of ***DirichletGroup(Abs(N),K)*** (with ***K*** a finite field of characteristic different from 2) which corresponds to the Legendre symbol $p \mapsto \left(\frac{\pm N}{p}\right)$. If ***N*** is 1 mod 4 the sign is $+1$, and -1 otherwise.

intrinsic DihedralForms (N :: RngIntElt :
ListOfPrimes := [], bound := 100, odd_only := true, quad_disc := 0,
completely_split := true, all_conjugacy_classes := true) -> Rec
This function computes all modular forms (in the sense of Section 2.1) of level ***N*** and weight p

over a finite field of characteristic p that come from dihedral representations whose determinant is the Legendre symbol of the quadratic field $K = \mathbb{Q}(\sqrt{\pm \textbf{\textit{quad\_disc}}})$ and which are obtained by induction of an unramified character of K. If ***quad_disc*** is 1 mod 4 the sign is $+1$, and -1 otherwise. If ***quad_disc*** is 0, the value of ***N*** is used. If the option ***completely_split*** is set, only those representations are returned which are completely split at p. If the option ***ListOfPrimes*** is assigned a non-empty list of primes, only those primes are considered as the characteristic. If it is the empty set, all primes p up to the ***bound*** are taken into consideration. If the option ***odd_only*** is true, only odd Galois representations are returned. If the option ***all_conjugacy_classes*** is true, each unramified character as above up to Galois conjugacy and up to taking inverses is used. Otherwise, a single choice is made. That there may be non-conjugate characters cutting out the same number field is due to the fact that there may be non-conjugate elements of the same order in the multiplicative group of a finite field.

2.3 Icosahedral modular forms

Eigenforms whose attached Galois representations take the group A_5 as projective images are called *icosahedral*. Since extensive tables of A_5-extensions of the rationals are available, one can consider icosahedral Galois representations which one knows very well. That allows one to test certain conjectures concerning modular forms on icosahedral ones.

We note the isomorphism $A_5 \cong \mathrm{SL}_2(\mathbb{F}_4)$. Thus, A_5-extentions of the rationals give rise to icosahedral Galois representations in characteristic 2 which (should) come from modular forms mod 2. It would also be possible to use certain other primes, but this has not been implemented.

intrinsic A5Form (f :: RngUPolElt) -> Rec
Returns the icosahedral form in characteristic 2 and weight 2 of smallest predicted level corresponding to the polynomial ***f*** which is expected to be of degree 5 and whose Galois group is supposed to be A_5. No checks about ***f*** are performed.

2.4 The Hecke algebra format

The data concerning the Hecke algebra of an eigenform that is computed by the function ***HeckeAlgebras*** is a record of the following form.
AlgebraData := recformat <

Level	***: RngIntElt,***
Weight	***: RngIntElt,***
Characteristic	***: RngIntElt,***
BaseFieldDegree	***: RngIntElt,***
CharacterOrder	***: RngIntElt,***
CharacterConductor	***: RngIntElt,***
CharacterIndex	***: RngIntElt,***
AlgebraFieldDegree	***: RngIntElt,***
ResidueDegree	***: RngIntElt,***
Dimension	***: RngIntElt,***
GorensteinDefect	***: RngIntElt,***
EmbeddingDimension	***: RngIntElt,***

NilpotencyOrder ***: RngIntElt,***
Relations ***: Tup,***
NumberGenUsed ***: RngIntElt,***
ImageName ***: MonStgElt,***
Polynomial ***: RngUPolElt***

>;

Level and ***Weight*** have the obvious meaning. Let K be the base field for the space of modular symbols used. It is (expected to be) a finite field. Then ***Characteristic*** is the characteristic of K and ***BaseFieldDegree*** is the degree of K over its prime field. The entries ***CharacterOrder***, ***CharacterConductor*** and ***CharacterIndex*** concern the Dirichlet character for which the modular symbols have been computed. The latter field is the index of the character in ***Elements(DirichletGroup(·))***. Note that that might change between different versions of MAGMA. The fields ***ResidueDegree*** (over the prime field), ***Dimension*** and ***GorensteinDefect*** have their obvious meaning for the Hecke algebra in question. The tuple

<AlgebraFieldDegree, EmbeddingDimension, NilpotencyOrder, Relations>

are data from which ***AffineAlgebra*** can recreate the Hecke algebra up to isomorphism. ***NumberGenUsed*** indicates the number of generators used by the package for the computation of the Hecke algebra. This number is usually much smaller than the Sturm bound. ***ImageName*** and ***Polynomial*** have the same meaning as in the record ***ModularFormFormat***.

2.5 Hecke algebras

intrinsic HeckeAlgebras (eps :: GrpDrchElt, weight :: RngIntElt :
UserBound := 0, first_test := 3, test_interval := 1, when_test_p := 3,
when_test_bad := 4, test_sequence := [], dimension_factor := 2,
ms_space := 0, cuspidal := true, DegreeBound := 0, OperatorList := [],
over_residue_field := true, try_minimal := true, force_local := false,
) -> SeqEnum, SeqEnum, ModSym, Tup, Tup
intrinsic HeckeAlgebras (t :: Rec :
UserBound := 0, first_test := 3, test_interval := 1, when_test_p := 3,
when_test_bad := 4, test_sequence := [], dimension_factor := 2,
ms_space := 0, cuspidal := true, DegreeBound := 0, OperatorList := [],
over_residue_field := true, try_minimal := true, force_local := false,
) -> SeqEnum, SeqEnum, ModSym, Tup, Tup

These functions compute all local Hecke algebras (up to Galois conjugacy) in the specified ***weight*** for the given Dirichlet character ***eps***, respectively those corresponding to the modular form ***t*** given by a record of type ***ModularFormFormat***. The functions return 5 values ***A,B,C,D,E***. ***A*** contains a list of records of type ***AlgebraData*** describing the local Hecke algebra factors. ***B*** is a list containing the local Hecke algebra factors as matrix algebras. ***C*** is the space of modular symbols used in the computations. ***D*** is a tuple containing the base change tuples describing the local Hecke factors. We need to know ***D*** in order to compute matrices representing

Hecke operators for the local factor. Finally, ***E*** contains a tuple consisting of all Hecke operators computed so far for each local factor of the Hecke algebra.

The usage in practice is described in the example at the beginning of this manual. We now explain the different options in detail.

The modular symbols space to be used in the computations can be determined as follows. The option ***ms_space*** can be set to the values 1 (the plus-space), -1 (the minus-space) and 0 (the full space). Whether the restriction to the cuspidal subspace is taken, is determined by ***cuspidal***. It is not necessary to pass to the cuspidal subspace, for example, if a cusp form is given by a coefficient function (see the description of the record ***ModularFormFormat***).

In some cases, a list of Hecke operators on the modular symbols space in question may already have been computed. In order to prevent MAGMA from redoing their computations, they may be passed on to the function using the option ***OperatorList***.

Often, one wants to compute the local Hecke algebra of a modular form whose degree of the coefficient field over its prime field is known, e.g. in the case of an icosahedral form in characteristic 2 for the trivial Dirichlet character the coefficient field is $\mathbb{F}_4$. By assigning a positive value to the option ***DegreeBound*** the function will automatically discard any systems of eigenvalues beyond that bound, which speeds up the computations. One must be a bit careful with this option, as there may be cases when the bound may not be respected at "bad primes". But it usually suffices to take twice the degree of the coefficient field, e.g. one chooses ***DegreeBound := 4*** in the icosahedral example just described. If no system of eigenvalues should be discarded for degree reasons, one must set ***DegreeBound := 0***.

All of the options ***first_test***, ***test_interval***, ***when_test_p***,***when_test_bad***, ***test_sequence***, ***force_local***, ***dimension_factor*** and ***UserBound*** concern the stop criterion. Theoretically, the Sturm bound (see ***HeckeBound***) tells us up to which bound Hecke operators must be computed in order to be sure that they generate the whole Hecke algebra. In practice, however, the algorithm can often determine itself when enough Hecke operators have been computed to generate the algebra. That number is usually much smaller than the Sturm bound. The Sturm bound can be overwritten by assigning a positive number to ***UserBound***.

The stop criterion is the following. Let M be the modular symbols space used and S the set of Hecke operators computed so far. Then $M = \bigoplus_{i=1}^{r} M_i$ (for some r) such that each M_i is respected by the Hecke operators and the minimal polynomial of each $T \in S$ restricted to M_i is a power of an irreducible polynomial (i.e. each M_i is a primary space for the action of the algebra generated by all elements of S). Let A_i be the algebra generated by $T|_{M_i}$ for all $T \in S$. One knows (in many cases, and in all cases of interest) that A_i is equal to a direct product of local Hecke algebras if one has the equality

$$f \times \dim(A_i) = \text{ dimension of } M_i.$$

Here, f is given by ***dimension_factor*** and should be 1 if the plus-space or the minus space of modular symbols are used, and 2 otherwise. The correct assignment of ***dimension_factor*** must be made by hand, whence experimentations are possible. If the stop criterion is not reached, the algorithm terminates at the Hecke bound.

It may happen that, when the stop criterion is reached, one A_i is isomorphic to a direct product of more than one local Hecke algebras. If in that case the option ***force_local*** is ***true***, the computation of Hecke operators is continued until each A_i is isomorphic to a single Hecke factor. If

force_local is ***false***, then a fast localisation algorithm is applied to each A_i. The option is useful, when one expects only a single local Hecke algebra factor, for example, when a modular form is given.

In many cases of interest the Hecke operator T_p with p the characteristic is needed in order to generate the whole Hecke algebra. The option ***when_test_p*** tells the algorithm at which step to compute T_p. It is very advisable to choose a small number. In practice, the stop criterion is reached after very few steps, e.g. 5 steps, when T_p is computed early. Otherwise, the algorithm often has to continue until T_p is computed, although most of the operators before did not change the generated algebra. The option ***when_test_bad*** has a similar meaning for the T_l for primes l dividing the level. However, paying attention to them is only required when the modular form is old at l. Moreover, one can assign a list of primes to ***test_sequence***. The algorithm will then start with the Hecke operators indicated by that sequence, and then continue with the others.

The option ***first_test*** tells the algorithm at which step the first test for the stop criterion is to be performed. The next test is then carried out after ***test_interval*** many steps, and so on. These numbers should be chosen small, too, unless the dimension test takes much time, which is rare, so that one wants to perform it less often, meaning that possibly more Hecke operators than necessary are computed (time consuming).

The option ***over_residue_field*** tells the algorithm whether at the end of the computation the local Hecke factors should be base changed to their residue field. If that is done, only one of the conjugate local factors of the base changed algebra is retained.

Finally, the option ***try_minimal*** is passed on to ***AffineAlgebra***, when the output is generated. Calling that function with the option set ***true*** can sometimes be very time consuming, but makes the output much shorter.

2.6 Storage functions

The package provides functions to store a list whose elements are records of type ***AlgebraData*** in a file, and to re-read it. The usage of these functions is explained in the example at the beginning of this manual.

intrinsic CreateStorageFile (filename :: MonStgElt)
This function prepares the file ***filename*** for storing the data.

intrinsic StoreData (filename :: MonStgElt, forms :: SeqEnum)
This functions appends the list ***forms*** of Hecke algebra data to the file ***filename***. That file must have been created by ***CreateStorageFile***.

intrinsic StoreData (filename :: MonStgElt, form :: Rec)
This function appends the Hecke algebra data ***form*** to the file ***filename***. That file must have been created by ***CreateStorageFile***.

intrinsic RecoverData (LoadIn :: SeqEnum, LoadInRel :: Tup) -> SeqEnum
In order to read Hecke algebra data from file ***"name"***, proceed as follows:

> load "name";

> readData := RecoverData(LoadIn,LoadInRel).

Then ***readData*** will contain a list whose elements are records of type ***AlgebraData***.

2.7 Output functions

intrinsic HeckeAlgebraPrint (ha :: SeqEnum)
intrinsic HeckeAlgebraPrint1 (ha :: SeqEnum)
These functions print part of the data stored in the list ***ha*** of records of type ***AlgebraData*** in a human readable format.

intrinsic GetLevel (a :: Rec) -> Any
intrinsic GetWeight (a :: Rec) -> Any
intrinsic GetCharacteristic (a :: Rec) -> Any
intrinsic GetResidueDegree (a :: Rec) -> Any
intrinsic GetDimension (a :: Rec) -> Any
intrinsic GetGorensteinDefect (a :: Rec) -> Any
intrinsic GetEmbeddingDimension (a :: Rec) -> Any
intrinsic GetNilpotencyOrder (a :: Rec) -> Any
intrinsic GetHeckeBound (a :: Rec) -> Any
intrinsic GetPrimesUpToHeckeBound (a :: Rec) -> Any
intrinsic GetNumberOperatorsUsed (a :: Rec) -> Any
intrinsic GetPolynomial (a :: Rec) -> Any
intrinsic GetImageName (a :: Rec) -> Any
These functions return the property of the record ***a*** of type ***AlgebraData*** specified by the name of the function. If the corresponding attribute is not assigned, the empty string is returned.

intrinsic HeckeAlgebraLaTeX (ha :: SeqEnum, filename :: MonStgElt : which := [
<GetLevel,"Level">, <GetWeight,"Wt">, <GetResidueDegree,"ResD">,
<GetDimension,"Dim">, <GetEmbeddingDimension,"EmbDim">,
<GetNilpotencyOrder,"NilO">, <GetGorensteinDefect,"GorDef">,
<GetNumberOperatorsUsed,"#Ops">,
<GetPrimesUpToHeckeBound,"#(p<HB)">, <GetImageName,"Gp">])
This function creates the LaTeX file ***filename*** containing a longtable consisting of certain properties of the objects in ***ha*** which are supposed to be records of type ***AlgebraData***. The properties to be written are indicated by the list given in the option ***which*** consisting of tuples ***<f, name>***. Here ***f*** is a function that evaluates a record of type ***AlgebraData*** to some Magma object which is afterwards transformed into a string using ***Sprint***. Examples for ***f*** are the functions ***GetLevel*** etc., which are described above. The ***name*** will appear in the table header. For a sample usage, see the example at the beginning of this manual.

2.8 Other functions

intrinsic HeckeBound (N :: RngIntElt, k :: RngIntElt) -> RngIntElt
intrinsic HeckeBound (eps :: GrpDrchElt, k :: RngIntElt) -> RngIntElt
These functions compute the Hecke bound for weight ***k*** and level ***N***, respectively Dirichlet character ***eps***. Note that the Hecke bound is also often called the Sturm bound.

3 Algebra handling

3.1 Affine algebras

Let A be a commutative local Artin algebra with maximal ideal $\mathfrak{m}$ over a finite field k. The residue field $K = A/\mathfrak{m}$ is a finite extension of k. By base changing to K and taking one of the conjugate local factors, we now assume that $k = K$. The *embedding dimension* e is the k-dimension of $\mathfrak{m}/\mathfrak{m}^2$. By Nakayama's Lemma, this is the minimal number of generators for $\mathfrak{m}$. The name comes from the fact that there is a surjection

$$\pi : k[x_1, \ldots, x_e] \twoheadrightarrow A.$$

Its kernel is called the *relations ideal*. By the *nilpotency order* we mean the maximal integer n such that m^n is not the zero ideal. (As the algebra is local and Artin, its maximal ideal is nilpotent.) We know that the ideal

$$J^{n+1} \text{ with } J := (x_1, \ldots, x_e)$$

is in the kernel of π. So, in order to store π, we only need to store the kernel R of the linear map between two finite dimensional k-vector spaces

$$\pi_1 : k[x_1, \ldots, x_e]/J^{n+1} \twoheadrightarrow A.$$

From the tuple $< k, e, n, R >$ the algebra can be recreated (up to isomorphism). Let us point out, however, that from the tuple it is not obvious whether two algebras are isomorphic. That would have to be tested after recreating the algebras.

These functions are used in order to store the Hecke algebras computed by ***HeckeAlgebras*** in a way that does not use much memory, but retains the algebra up to isomorphism.

intrinsic AffineAlgebra (A :: AlgMat : try_minimal := true) -> RngMPolRes
intrinsic AffineAlgebra (A :: AlgAss : try_minimal := true) -> RngMPolRes
This function turns the local commutative algebra ***A*** into an affine algebra over its residue field. In fact, the algebra is first base changed to its residue field, then for one of the conjugate local factors an affine presentation is computed. If the option ***try_minimal*** is true, the number of relations will in general be smaller, but the computation time may be longer.

intrinsic AffineAlgebraTup (A :: AlgMat : try_minimal := true) -> Tup
intrinsic AffineAlgebraTup (A :: AlgAss : try_minimal := true) -> Tup
Given a commutative local Artin algebra ***A***, this function returns a tuple ***<k,e,n,R>***, consisting of the residue field ***k*** of ***A***, the embedding dimension ***e***, the nilpotency order ***n*** and relations ***R***. From these data, an affine algebra can be recreated which is isomorphic to one of the local factors of A base changed to its residue field. If the option ***try_minimal*** is true, the number of relations will in general be smaller, but the computation time may be longer.

intrinsic AffineAlgebra (form :: Rec) -> RngMPolRes
Given a record of type ***AlgebraData***, this function returns the corresponding Hecke algebra as an affine algebra.

intrinsic AffineAlgebra (A :: Tup) -> RngMPolRes
This function turns a tuple ***<k,e,n,R>***, as above consisting of a field ***k***, two integers ***e***, ***n*** (the embedding dimension and the nilpotency order) and relations ***R***, into an affine algebra.

3.2 Matrix algebra functions

intrinsic MatrixAlgebra (L :: SeqEnum) -> AlgMat
Given a list of matrices ***L***, this function returns the matrix algebra generated by the members of ***L***.

intrinsic RegularRepresentation (A :: AlgMat) -> AlgMat
This function computes the regular representation of the commutative matrix algebra ***A***.

intrinsic CommonLowerTriangular (A :: AlgMat) -> AlgMat
Given a local commutative matrix algebra ***A***, this function returns an isomorphic matrix algebra whose matrices are all lower triangular, after a scalar extension to the residue field and taking one of the Galois conjugate factors.

Base change

intrinsic BaseChange (S :: Tup, T :: Tup) -> Tup
This function computes the composition of the base change matrices ***T = <C,D>***, followed by those in ***S = <E,F>***.

intrinsic BaseChange (M :: Mtrx, T :: Tup) -> Mtrx
Given a matrix ***M*** and a tuple ***T = <C,D>*** of base change matrices (for a subspace), this function computes the matrix of ***M*** with respect to the basis corresponding to ***T***.

intrinsic BaseChange (M :: AlgMat, T :: Tup) -> AlgMat
Given a matrix algebra ***M*** and a tuple ***T = <C,D>*** of base change matrices (for a subspace), this function computes the matrix algebra of ***M*** with respect to the basis corresponding to ***T***.

Decomposition

intrinsic Decomposition (M :: Mtrx : DegBound := 0) -> Tup
intrinsic DecompositionUpToConjugation (M :: Mtrx : DegBound := 0) -> Tup
Given a matrix ***M***, these functions compute a decomposition of the standard vector space such that ***M*** acts as multiplication by a scalar on each summand. The output is a tuple consisting of base change tuples ***<C,D>*** corresponding to the summands. With the second usage, summands conjugate under the absolute Galois group only appear once.

intrinsic Decomposition (L :: SeqEnum : DegBound := 0) -> Tup
intrinsic DecompositionUpToConjugation (L :: SeqEnum : DegBound := 0) -> Tup
Given a sequence ***L*** of commuting matrices, these functions compute a decomposition of the standard vector space such that each matrix in ***L*** acts as multiplication by a scalar on each summand. The output is a tuple consisting of base change tuples ***<C,D>*** corresponding to the summands. With the second usage, summands conjugate under the absolute Galois group only appear once.

intrinsic Decomposition (A :: AlgMat : DegBound := 0) -> Tup
intrinsic DecompositionUpToConjugation (A :: AlgMat : DegBound := 0) -> Tup
Given a commutative matrix algebra ***A***, these functions compute a decomposition of the standard vector space such that each element in ***A*** acts as multiplication by a scalar on each summand. The output is a tuple consisting of base change tuples ***<C,D>*** corresponding to the summands. With the second usage, summands conjugate under the absolute Galois group only appear once.

intrinsic AlgebraDecomposition (A :: AlgMat : DegBound := 0) -> SeqEnum
intrinsic AlgebraDecompositionUpToConjugation (A :: AlgMat : DegBound := 0)
-> SeqEnum
Given a matrix algebra $\boldsymbol{A}$ over a finite field, these functions return a local factor of $\boldsymbol{A}$ after scalar extension to the residue field. With the second usage, factors conjugate under the absolute Galois group only appear once.

intrinsic ChangeToResidueField (A :: AlgMat) -> SeqEnum
This function is identical to ***AlgebraDecompositionUpToConjugation***.

Localisations

intrinsic Localisations (L :: SeqEnum) -> Tup, Tup
intrinsic Localisations (A :: AlgMat) -> Tup, Tup
Given a list $\boldsymbol{L}$ of commuting matrices or a commutative matrix algebra $\boldsymbol{A}$, this function computes two tuples $\boldsymbol{C}$, $\boldsymbol{D}$, where $\boldsymbol{C}$ contains a tuple consisting of the localisations of $\boldsymbol{A}$, respectively of the matrix algebra generated by $\boldsymbol{L}$, and $\boldsymbol{D}$ consists of the corresponding base change tuples.

3.3 Associative algebras

intrinsic Localisations (A :: AlgAss) -> SeqEnum
This function returns a list of all localisations of the Artin algebra $\boldsymbol{A}$, which is assumed to be commutative. The output is a list of associative algebras.

3.4 Gorenstein defect

Let A be a local Artin algebra over a field with unique maximal ideal $\mathfrak{m}$. We define the *Gorenstein defect* of A to be $(\dim_{A/\mathfrak{m}} A[\mathfrak{m}]) - 1$, which is equal to the number of A-module generators of the annihilator of the maximal ideal minus one. The algebra is said to be *Gorenstein* if its Gorenstein defect is equal to 0.

intrinsic GorensteinDefect (A :: RngMPolRes) -> RngIntElt
intrinsic GorensteinDefect (A :: AlgAss) -> RngIntElt
intrinsic GorensteinDefect (A :: AlgMat) -> RngIntElt
These functions return the Gorenstein defect of the local commutative algebra $\boldsymbol{A}$.

intrinsic IsGorenstein (M :: RngMPolRes) -> BoolElt
intrinsic IsGorenstein (M :: AlgAss) -> BoolElt
intrinsic IsGorenstein (M :: AlgMat) -> BoolElt
These functions test whether the commutative local algebra $\boldsymbol{M}$ is Gorenstein.