GAP: Weyl Modules

Release 1.0
30 March 2009

Documentation

S.R. Doty

http://doty.math.luc.edu/weylmodules

Acknowledgement

The development of this software was initiated in June 2003 while the author was visiting the Department of Pure Mathematics and Mathematical Statistics (DPMMS) at the University of Cambridge, and continued during subsequent visits to DPMMS in June 2004 and in May through July of 2007. The author was supported by a Yip Fellowship at Magdalene College, Cambridge in 2007.

The final stages of development took place in Chicago and in January 2009 at Universität Bielefeld, where the author was supported by a Mercator grant from the Deutsche Forschungsgemeinschaft (DFG).

The existence of this software owes much to the gentle prodding of Stuart Martin. Thanks are also due to Yutaka Yoshii for testing an earlier version of the software, and Matt Fayers for supplying his GAP code for computing the Mullineux map.

Contents

Copyright Notice

1 Weyl modules

This chapter discusses the commands available for computations with Weyl modules for a given semisimple simply-connected algebraic group G in positive characteristic p. Actually the group G itself never appears in any of the computations, which take place instead using the **algebra of distributions** (also known as the **hyperalgebra**) of G, taken over the prime field. One should refer to [Jan03] for the definition of the algebra of distributions, and other basic definitions and properties related to Weyl modules.

The algorithms are based on the method of [Irv86] (see also [Xi99]) and build on the existing Lie algebra functionality in GAP. In principle, one can work with arbitrary weights for an arbitrary (simple) root system; in practice, the functionality is limited by the size of the objects being computed. If your Weyl module has dimension in the thousands, you may have to wait a very long time for certain computations to finish.

The package is possibly most useful for doing computations in characteristic p where p is smaller than the Coxeter number. For such small primes, the general theory offers very little information.

Warning. At the core of many of the computations is a routine which produces a basis for the space of maximal vectors of a specified dominant weight in a Weyl module. Usually, that space has dimension at most 1. Cases for which there exist two or more independent maximal vectors of the same weight could possibly cause problems, so the code will emit an warning message if this occurs (and then try to continue). Such situations are relatively rare (and interesting); the smallest example known to the author occurs in Type D_4 in the Weyl module of highest weight [0,1,0,0], as pointed out in [CPS75], page 173. (I am grateful to Anton Cox for this reference.) See the examples in Section 1.6 to see the explicit form of the warning message.

1.1 Creating Weyl modules

There are two functions for creating a Weyl module.

1 ▶ `WeylModule( `*p*`, `*weight*`, `*type*`, `*rank*` )` F
▶ `WeylModule( `*V*`, `*weight*` )` F

The function `WeylModule` with four arguments creates a Weyl module over a field of characteristic p, of highest weight *weight*, for the root system of Type *type* and rank *rank*. In the second form, with two arguments, V is an existing Weyl module and the new Weyl module has the same characteristic and root system as the existing one.

```
gap> V:= WeylModule(3, [3,4], "A", 2);
<Type A2 Weyl module of highest weight [ 3, 4 ] at prime p = 3>
gap> W:= WeylModule(V, [3,0]);
<Type A2 Weyl module of highest weight [ 3, 0 ] at prime p = 3>
```

There is also a category of Weyl modules.

2 ▶ `IsWeylModule( `*V*` )` C

returns `true` iff V belongs to the category.

```
gap> IsWeylModule(W);
true
```

1.2 Creating quotients of Weyl modules

Quotients of Weyl modules are also supported. They are created by the command

1 ▶ `QuotientWeylModule( V, list )` F

where V is an existing Weyl module and *list* is a list of basis vectors spanning a submodule of V. Usually one gets such a basis by running `SubWeylModule` (see Section 1.7 below).

```
gap> V:= WeylModule(2,[2,2],"B",2);
<Type B2 Weyl module of highest weight [ 2, 2 ] at prime p = 2>
gap> m:= MaximalVectors(V);
[ 1*v0, y1*v0, y2*v0, y1*y2*v0+y3*v0, y2*y3*v0, y1*y2*y3*v0, y1*y2*y3*y4*v0 ]
gap> sub:=SubWeylModule(V, m[7]);
[ y1*y2*y3*y4*v0 ]
gap> Q:= QuotientWeylModule(V, sub);
<Quotient of Type B2 Weyl module of highest weight [ 2, 2 ] at prime p = 2>
```

In the above example, we first created a Weyl module, then computed its maximal vectors. The last maximal vector generates a one dimensional submodule (a copy of the trivial module) and we formed the corresponding quotient Weyl module.

There is also a category of quotient Weyl modules.

2 ▶ `IsQuotientWeylModule( Q )` C

returns `true` iff Q belongs to the category.

```
gap> IsQuotientWeylModule(Q);
true
```

1.3 Basis, dimension, and other miscellaneous commands

Let V be a Weyl module, or a quotient Weyl module.

1 ▶ `TheLieAlgebra( V )` F
▶ `BasisVecs( V )` F
▶ `Dim( V )` F
▶ `Generator( V )` F
▶ `TheCharacteristic( V )` F

These commands return the underlying Lie algebra associated to V, a basis (of weight vectors) for V, the dimension of V, the standard generator of V, and the characteristic of the underlying field, respectively. In case V is a quotient Weyl module, `BasisVecs` returns a complete set of linearly independent coset representatives for the quotient.

```
gap> V:= WeylModule(2, [1,0], "G", 2);
<Type G2 Weyl module of highest weight [ 1, 0 ] at prime p = 2>
gap> TheLieAlgebra(V);
<Lie algebra of dimension 14 over Rationals>
gap> b:= BasisVecs(V);
[ 1*v0, y1*v0, y3*v0, y4*v0, y5*v0, y6*v0, y1*y6*v0 ]
gap> Dim(V);
7
gap> g:= Generator(V);
1*v0
gap> TheCharacteristic(V);
```

2

2 ▶ `ActOn( V, u, v )` F

`ActOn` returns the result of acting by a hyperalgebra element u on a vector v. Here v must be an element of V, where V is a Weyl module or a quotient Weyl module.

For example, with V as defined above in the preceding example, we have

```
gap> L:= TheLieAlgebra(V);
<Lie algebra of dimension 14 over Rationals>
gap> b:= BasisVecs(V);
[ 1*v0, y1*v0, y3*v0, y4*v0, y5*v0, y6*v0, y1*y6*v0 ]
gap> g:= LatticeGeneratorsInUEA(L);
[ y1, y2, y3, y4, y5, y6, x1, x2, x3, x4, x5, x6, ( h13/1 ), ( h14/1 ) ]
gap> ActOn(V, g[1]^2 + g[7], b[1]);
0*v0
gap> ActOn(V, g[1]*g[6], b[1]);
y1*y6*v0
```

Note that the command `LatticeGeneratorsInUEA` is a pre-existing GAP command; see the chapter on Lie algebras in the GAP reference manual for further details. For our purposes, these elements are regarded as standard generators of the hyperalgebra.

1.4 Weight of a vector; weights of a list of vectors

One often wants to know the weight of a given vector in a Weyl module or a quotient Weyl module. Of course, it has to be a weight vector. The command

1 ▶ `Weight( v )` F

returns the weight of the weight vector v.

Another common situation is that one has a list *lst* of weight vectors (maybe a basis or a list of maximal vectors, or a basis of a submodule) and one wants to know the weight of each vector in the list. This is obtained by the command

2 ▶ `List( lst, Weight )` F

which maps the Weight function onto each element of the list *lst* in turn, making a list of the results.

```
gap> V:= WeylModule(2, [2,0], "A", 2);
<Type A2 Weyl module of highest weight [ 2, 0 ] at prime p = 2>
gap> b:= BasisVecs(V);
[ 1*v0, y1*v0, y3*v0, y1^(2)*v0, y1*y3*v0, y3^(2)*v0 ]
gap> List(b, Weight);
[ [ 2, 0 ], [ 0, 1 ], [ 1, -1 ], [ -2, 2 ], [ -1, 0 ], [ 0, -2 ] ]
gap> Weight( b[2] );
[ 0, 1 ]
gap> m:= MaximalVectors(V);
[ 1*v0, y1*v0 ]
gap> List(m, Weight);
[ [ 2, 0 ], [ 0, 1 ] ]
```

1.5 Structure of Weyl modules

One of the most useful commands is

1 ▶ `SubmoduleStructure( V )` F

which returns a complete list of primitive vectors in a Weyl module V, and along the way prints out an analysis of the submodule lattice structure of V. WARNING: If the dimension of V is large this can take a very long time.

```
gap> V:= WeylModule(3, [3,3], "A", 2);
<Type A2 Weyl module of highest weight [ 3, 3 ] at prime p = 3>
gap> v:= SubmoduleStructure(V);
Level 1
-maximal vector v1 = y1*y2*y3*v0+y1^(2)*y2^(2)*v0 of weight [ 1, 1 ]
Level 2
-maximal vector v2 = y1^(2)*y2*v0 of weight [ 0, 3 ]
-maximal vector v3 = -1*y1*y2^(2)*v0+y2*y3*v0 of weight [ 3, 0 ]
-primitive vector v4 = y1*y2*y3^(2)*v0 of weight [ 0, 0 ]
Level 3
-maximal vector v5 = y1*v0 of weight [ 1, 4 ]
-maximal vector v6 = y2*v0 of weight [ 4, 1 ]
Level 4
-primitive vector v7 = y1^(3)*y2^(3)*v0+y3^(3)*v0 of weight [ 0, 0 ]
Level 5
-maximal vector v8 = 1*v0 of weight [ 3, 3 ]
The submodule <v1> contains v1
The submodule <v2> contains v1 v2
The submodule <v3> contains v1 v3
The submodule <v4> contains v1 v4
The submodule <v5> contains v1 v2 v3 v4 v5
The submodule <v6> contains v1 v2 v3 v4 v6
The submodule <v7> contains v1 v2 v3 v4 v5 v6 v7
The submodule <v8> contains v1 v2 v3 v4 v5 v6 v7 v8
[ y1*y2*y3*v0+y1^(2)*y2^(2)*v0, y1^(2)*y2*v0, -1*y1*y2^(2)*v0+y2*y3*v0,
  y1*y2*y3^(2)*v0, y1*v0, y2*v0, y1^(3)*y2^(3)*v0+y3^(3)*v0, 1*v0 ]
```

This shows that V has eight primitive vectors, six of which are maximal. The submodule generated by each primitive vector is shown. The *levels* are the subquotient layers of the socle series of V, so this Weyl module has a simple socle of highest weight [1,1], there are two simple composition factors of highest weight [0,3] and [3,0] extending the socle, and so on. This example is treated in [DM08], where one can also find a diagram depicting the structure.

2 ▶ `SocleWeyl( V )` F

`SocleWeyl` returns a list of maximal vectors of the Weyl module V generating the socle of V. Note that V can also be a quotient Weyl module.

For example, with V as above, we have:

```
gap> SocleWeyl(V);
[ y1*y2*y3*v0+y1^(2)*y2^(2)*v0 ]
```

3 ▶ `ExtWeyl( V, list )` F

`ExtWeyl` returns a list of maximal vectors generating the socle of the quotient V/S where S is the submodule of V generated by the vectors in the given *list*.

For example, with V as above, we have:

```
gap> soc:= SocleWeyl(V);
[ y1*y2*y3*v0+y1^(2)*y2^(2)*v0 ]
gap> ExtWeyl(V, soc);
[ y1^(2)*y2*v0, -1*y1*y2^(2)*v0+y2*y3*v0, y1*y2*y3^(2)*v0 ]
```

Maximal vectors generating extensions are only determined modulo S. In some cases, the maximal vectors returned by `ExtWeyl` are not the same as the maximal vectors returned by `SocleWeyl(`V/S`)`. In order to properly detect the splitting of some extension groups, the `ExtWeyl` function may in some cases replace a maximal vector by a different choice of representative. For example, the Weyl module of highest weight [2,0] in characteristic 2 for Type G_2 exhibits such a difference:

```
gap> V:= WeylModule(2,[2,0],"G",2);
<Type G2 Weyl module of highest weight [ 2, 0 ] at prime p = 2>
gap> S:= SocleWeyl(V);
[ y1*v0, y4*v0 ]
gap> e:= ExtWeyl(V,S);
[ y1*y6*v0+y3*y5*v0+y4^(2)*v0 ]
gap> SocleWeyl( QuotientWeylModule(V,SubWeylModule(V,S)) );
[ y4^(2)*v0 ]
```

In this case, the vector $y_1 y_6 v_0 + y_3 y_5 v_0 + y_4^{(2)} v_0$ is a better choice of generator than $y_4^{(2)} v_0$. Both choices are equivalent module the submodule generated by the socle, but the first choice reveals that only one of the socle factors is extended, as one sees by running `SubmoduleStructure` on this module. (I am indebted to Yutaka Yoshii for finding this example and pointing out some bugs in an earlier version of the GAP code.)

4 ▶ `MaximalSubmodule(` V `)` F

`MaximalSubmodule` returns a basis of weight vectors for the maximal submodule of a given Weyl module V. The corresponding quotient is irreducible, of the same highest weight as V.

```
gap> V:= WeylModule(2, [4,0], "A", 2);
<Type A2 Weyl module of highest weight [ 4, 0 ] at prime p = 2>
gap> Dim(V);
15
gap> max:= MaximalSubmodule(V);
[ y1*v0, y3*v0, y1*y3*v0, y1^(3)*v0, y1*y3^(2)*v0, y1^(2)*y3*v0, y3^(3)*v0,
  y1^(3)*y3*v0, y1*y3^(3)*v0, y1^(2)*v0, y3^(2)*v0, y1^(2)*y3^(2)*v0 ]
gap> Length(max);
12
gap> Q:= QuotientWeylModule(V, max);
<Quotient of Type A2 Weyl module of highest weight [ 4, 0 ] at prime p = 2>
gap> b:= BasisVecs(Q);
[ 1*v0, y1^(4)*v0, y3^(4)*v0 ]
gap> List(b, Weight);
[ [ 4, 0 ], [ -4, 4 ], [ 0, -4 ] ]
```

1.6 Maximal and primitive vectors; homomorphisms between Weyl modules

A **maximal vector** is by definition a non-zero vector killed by the action of the unipotent radical of the positive Borel subgroup; see [Jan03] for further details. If V is a Weyl module, or a quotient Weyl module, the commands

1 ▶ `MaximalVectors( V )` F
▶ `MaximalVectors( V, weight )` F

respectively return a list of linearly independent vectors in V spanning the subspace of all maximal vectors of V, or a list of linearly independent vectors spanning the subspace of maximal vectors of the given weight space. (Note that linear combinations of maximal vectors are again maximal.)

In case V is a Weyl module, each maximal vector of V corresponds to a nontrivial homomorphism from the Weyl module of that highest weight into V. Hence the above commands can be used to determine the space $\mathrm{Hom}(W, V)$ for two given Weyl modules W, V.

```
gap> V:= WeylModule(2, [2,2,2], "A", 3);
<Type A3 Weyl module of highest weight [ 2, 2, 2 ] at prime p = 2>
gap> m:= MaximalVectors(V);
[ 1*v0, y1*v0, y2*v0, y3*v0, y1*y3*v0, y1*y2*y3*v0+y3*y4*v0+y6*v0,
  y1*y2*y5*v0+y2*y3*y4*v0+y4*y5*v0, y1*y2*y4*v0, y2*y3*y5*v0,
  y1*y2*y3*y4*v0+y1*y4*y5*v0, y1*y2*y3*y5*v0, y1*y2*y3*y4*y5*y6*v0 ]
gap> m:= MaximalVectors(V, [0,3,2]);
[ y1*v0 ]
```

Here are two examples where the space of maximal vectors for a specified weight has dimension strictly greater than 1. As mentioned at the beginning of the chapter, such examples generate a warning message (which is safe to ignore in the two cases given below).

```
gap> V:= WeylModule(2, [0,1,0,0], "D", 4);
<Type D4 Weyl module of highest weight [ 0, 1, 0, 0 ] at prime p = 2>
gap> m:= MaximalVectors(V);
*******************************************************
** WARNING! Dimension > 1 detected
** in maximal vecs of weight [ 0, 0, 0, 0 ]
** in Weyl module of highest weight
** [ 0, 1, 0, 0 ]
*******************************************************
[ 1*v0, y5*y10*v0+y6*y9*v0, y2*y11*v0+y5*y10*v0+y12*v0 ]
gap> List(m, Weight);
[ [ 0, 1, 0, 0 ], [ 0, 0, 0, 0 ], [ 0, 0, 0, 0 ] ]
gap> V:= WeylModule(2, [0,1,0,0,0,0], "D", 6);
<Type D6 Weyl module of highest weight [ 0, 1, 0, 0, 0, 0 ] at prime p = 2>
gap> m:= MaximalVectors(V);
*******************************************************
** WARNING! Dimension > 1 detected
** in maximal vecs of weight [ 0, 0, 0, 0, 0, 0 ]
** in Weyl module of highest weight
** [ 0, 1, 0, 0, 0, 0 ]
*******************************************************
[ 1*v0, y7*y28*v0+y8*y27*v0+y13*y25*v0+y18*y22*v0, y2*y29*v0+y7*y28*v0+y30*v0
 ]
gap> List(m, Weight);
[ [ 0, 1, 0, 0, 0, 0 ], [ 0, 0, 0, 0, 0, 0 ], [ 0, 0, 0, 0, 0, 0 ] ]
```

Given a weight vector v in a Weyl module, or quotient Weyl module, one can test whether or not the vector v is maximal. The two forms of this command are:

2 ▶ `IsMaximalVector( V, v )` F
▶ `IsMaximalVector( V, lst, v )` F

and in the second form *lst* must be a basis of weight vectors for a submodule of V. The first form of the command returns `true` iff v is maximal in V; the second form returns `true` iff the image of v is maximal in the quotient V/S where S is the submodule spanned by *lst*.

If V is a Weyl module, a **primitive vector** in V is a vector whose image in some sub-quotient is maximal (see [Xi99]). Maximal vectors are always primitive, by definition. Clearly, the (independent) primitive vectors are in bijective correspondence with the composition factors of V.

If V is a Weyl module, the command

3 ▶ `PrimitiveVectors( V )` F

returns a list of the primitive vectors of V. This is the same list returned by `SubmoduleStructure` but it should execute faster since it does not bother about computing structure. For example:

```
gap> V:= WeylModule(3, [3,3], "A", 2);
<Type A2 Weyl module of highest weight [ 3, 3 ] at prime p = 3>
gap> p:= PrimitiveVectors(V);
[ y1*y2*y3*v0+y1^(2)*y2^(2)*v0, y1^(2)*y2*v0, -1*y1*y2^(2)*v0+y2*y3*v0,
  y1*y2*y3^(2)*v0, y1*v0, y2*v0, y1^(3)*y2^(3)*v0+y3^(3)*v0, 1*v0 ]
```

WARNING: If the dimension of V is large, this command can take a very long time to execute.

1.7 Submodules

Given a vector v or a list *lst* of vectors, in a given Weyl module or quotient Weyl module, V, one obtains a basis of weight vectors for the submodule of V generated by v or *lst* by the appropriate command listed below.

1 ▶ `SubWeylModule( V, v )` F
▶ `SubWeylModule( V, lst )` F

WARNING: This can take a very long time, if the dimension of V is large.

Here is an example, in which we find a submodule and compute the corresponding quotient of the Weyl module:

```
gap> V:= WeylModule(2, [8,0], "A", 2);
<Type A2 Weyl module of highest weight [ 8, 0 ] at prime p = 2>
gap> m:= MaximalVectors(V);
[ 1*v0, y1*v0, y1^(3)*y3^(2)*v0 ]
gap> List(m, Weight);
[ [ 8, 0 ], [ 6, 1 ], [ 0, 1 ] ]
gap> s:= SubWeylModule(V, m[2]);
[ y1*v0, y3*v0, y1*y3*v0, y1^(3)*v0, y1*y3^(2)*v0, y1^(5)*v0, y1*y3^(4)*v0,
  y1^(2)*y3*v0, y3^(3)*v0, y1^(4)*y3*v0, y3^(5)*v0, y1^(3)*y3*v0,
  y1*y3^(3)*v0, y1^(5)*y3*v0, y1*y3^(5)*v0, y1^(3)*y3^(2)*v0, y1^(7)*v0,
  y1^(3)*y3^(4)*v0, y1^(5)*y3^(2)*v0, y1*y3^(6)*v0, y1^(2)*y3^(3)*v0,
  y1^(6)*y3*v0, y1^(2)*y3^(5)*v0, y1^(4)*y3^(3)*v0, y3^(7)*v0,
  y1^(3)*y3^(3)*v0, y1^(7)*y3*v0, y1^(3)*y3^(5)*v0, y1^(5)*y3^(3)*v0,
  y1*y3^(7)*v0 ]
```

```
gap> Q:= QuotientWeylModule(V, s);
<Quotient of Type A2 Weyl module of highest weight [ 8, 0 ] at prime p = 2>
gap> BasisVecs(Q);
[ 1*v0, y1^(2)*v0, y3^(2)*v0, y1^(4)*v0, y1^(2)*y3^(2)*v0, y1^(6)*v0,
  y3^(4)*v0, y1^(4)*y3^(2)*v0, y1^(8)*v0, y1^(2)*y3^(4)*v0, y1^(6)*y3^(2)*v0,
  y3^(6)*v0, y1^(4)*y3^(4)*v0, y1^(2)*y3^(6)*v0, y3^(8)*v0 ]
gap> Dim(Q);
15
```

One can also construct sub-quotients (continuing the preceding computation):

```
gap> mm:= MaximalVectors(Q);
[ 1*v0, y1^(2)*v0 ]
gap> subq:= SubWeylModule(Q, mm[2]);
[ y1^(2)*v0, y3^(2)*v0, y1^(2)*y3^(2)*v0, y1^(6)*v0, y1^(2)*y3^(4)*v0,
  y1^(4)*y3^(2)*v0, y3^(6)*v0, y1^(6)*y3^(2)*v0, y1^(2)*y3^(6)*v0 ]
gap> List(subq, Weight);
[ [ 4, 2 ], [ 6, -2 ], [ 2, 0 ], [ -4, 6 ], [ 0, -2 ], [ -2, 2 ], [ 2, -6 ],
  [ -6, 4 ], [ -2, -4 ] ]
```

Here, we have constructed a basis of weight vectors for the simple socle of the quotient `Q`.

Given a Weyl module, or a quotient Weyl module, V, a list *lst* of weight vectors forming a basis for a submodule, and a vector v, the command

2 ▶ `IsWithin(` V`,` *lst*`,` v `)`

returns `true` iff the given vector v lies within the submodule given by the basis *lst*.

1.8 Weights and weight spaces

If V is a Weyl module, or a quotient Weyl module, the following commands are available.

1 ▶ `Weights(` V `)` F
▶ `DominantWeights(` V `)` F
▶ `WeightSpaces(` V `)` F
▶ `DominantWeightSpaces(` V `)` F
▶ `WeightSpace(` V`,` *weight* `)` F

`Weights` returns a list of the weights of V, without multiplicities; `DominantWeights` returns a list of the dominant weights of V, again without multiplicities.

`WeightSpaces` returns a list consisting of each weight followed by a basis of the corresponding weight space; `DominantWeightSpaces` returns just the sublist containing the dominant weights and the corresponding weight spaces.

Finally, `WeightSpace` returns a basis of the particular weight space given by the specified *weight*.

```
gap> V:= WeylModule(2, [1,0,0], "A", 3);
<Type A3 Weyl module of highest weight [ 1, 0, 0 ] at prime p = 2>
gap> Weights(V);
[ [ 1, 0, 0 ], [ -1, 1, 0 ], [ 0, -1, 1 ], [ 0, 0, -1 ] ]
gap> DominantWeights(V);
[ [ 1, 0, 0 ] ]
gap> WeightSpaces(V);
[ [ 1, 0, 0 ], [ 1*v0 ], [ -1, 1, 0 ], [ y1*v0 ], [ 0, -1, 1 ], [ y4*v0 ],
  [ 0, 0, -1 ], [ y6*v0 ] ]
gap> DominantWeightSpaces(V);
[ [ 1, 0, 0 ], [ 1*v0 ] ]
gap> WeightSpace(V, [-1,1,0]);
[ y1*v0 ]
gap> WeightSpace(V, [0,1,0]);
fail
```

The last command prints `fail` because there are no weight vectors of weight [0,1,0] in the indicated Weyl module.

2 Characters and decomposition numbers

(Formal) characters can be computed for Weyl modules and simple modules. In the latter case, this is done recursively using Steinberg's tensor product theorem; the characters of the simples of restricted highest weight are obtained by first computing the maximal submodule and then forming the corresponding quotient.

2.1 Characters

Given a Weyl module or quotient Weyl module V, the command:

1 ▶ `Character( V )` F

returns the (formal) character of V, as a list of weights and multiplicities (the multiplicity of each weight follows the weight in the list). For example,

```
gap> V:= WeylModule(3, [3,0], "A", 2);
<Type A2 Weyl module of highest weight [ 3, 0 ] at prime p = 3>
gap> Character(V);
[ [ 3, 0 ], 1, [ 1, 1 ], 1, [ 2, -1 ], 1, [ -1, 2 ], 1, [ 0, 0 ], 1,
  [ -3, 3 ], 1, [ 1, -2 ], 1, [ -2, 1 ], 1, [ -1, -1 ], 1, [ 0, -3 ], 1 ]
gap> S:= MaximalSubmodule(V);
[ y1*v0, 2*y1^(2)*v0, 2*y3*v0, y1*y3*v0, 2*y1^(2)*y3*v0, y3^(2)*v0,
  2*y1*y3^(2)*v0 ]
gap> Character( QuotientWeylModule(V, S) );
[ [ 3, 0 ], 1, [ -3, 3 ], 1, [ 0, -3 ], 1 ]
```

Of course, characters of Weyl modules are independent of the characteristic.

2 ▶ `SimpleCharacter( p, wt, t, r )` F
▶ `SimpleCharacter( V, wt )` F

In the first form, the command `SimpleCharacter` returns the character of the simple module of highest weight wt in characteristic p, for the root system of Type t and rank r. In the second form, V is an existing Weyl module and the data p, t, and r are taken from the same data used to define V.

```
gap> SimpleCharacter(3, [3,0], "A", 2);
[ [ 3, 0 ], 1, [ -3, 3 ], 1, [ 0, -3 ], 1 ]
```

3 ▶ `Character( lst )` F

returns the character of the submodule (of a Weyl module, or a quotient Weyl module) spanned by the independent weight vectors in the given *lst*.

```
gap> V:= WeylModule(2, [4,0], "A", 2);
<Type A2 Weyl module of highest weight [ 4, 0 ] at prime p = 2>
gap> m:= MaximalVectors(V);
[ 1*v0, y1*v0 ]
gap> simple:= SubWeylModule(V, m[2]);
[ y1*v0, y3*v0, y1*y3*v0, y1^(3)*v0, y1*y3^(2)*v0, y1^(2)*y3*v0, y3^(3)*v0,
  y1^(3)*y3*v0, y1*y3^(3)*v0 ]
gap> Character(simple);
[ [ 2, 1 ], 1, [ 3, -1 ], 1, [ 1, 0 ], 1, [ -2, 3 ], 1, [ 0, -1 ], 1,
  [ -1, 1 ], 1, [ 1, -3 ], 1, [ -3, 2 ], 1, [ -1, -2 ], 1 ]
```

In the preceding example, we obtain the character of the simple socle of the Type A_2 Weyl module of highest weight [4,0], in characteristic 2.

4 ► `DifferenceCharacter(` *c1*`,` *c2* `)` F

`DifferenceCharacter` returns the difference of two given characters, or `fail` if the difference is not another character. The arguments must be characters.

In the following example, we compute the character of the maximal submodule of the Weyl module of highest weight [6,0] for Type A_2 in characteristic 2.

```
gap> ch1:= Character( WeylModule(2, [6,0], "A", 2) );
[ [ 6, 0 ], 1, [ 4, 1 ], 1, [ 5, -1 ], 1, [ 2, 2 ], 1, [ 3, 0 ], 1, [ 0, 3 ],
  1, [ 4, -2 ], 1, [ 1, 1 ], 1, [ -2, 4 ], 1, [ 2, -1 ], 1, [ -1, 2 ], 1,
  [ -4, 5 ], 1, [ 3, -3 ], 1, [ 0, 0 ], 1, [ -3, 3 ], 1, [ -6, 6 ], 1,
  [ 1, -2 ], 1, [ -2, 1 ], 1, [ -5, 4 ], 1, [ 2, -4 ], 1, [ -1, -1 ], 1,
  [ -4, 2 ], 1, [ 0, -3 ], 1, [ -3, 0 ], 1, [ 1, -5 ], 1, [ -2, -2 ], 1,
  [ -1, -4 ], 1, [ 0, -6 ], 1 ]
gap> ch2:= SimpleCharacter(2, [6,0], "A", 2);
[ [ 6, 0 ], 1, [ -2, 4 ], 1, [ 2, -4 ], 1, [ 2, 2 ], 1, [ -6, 6 ], 1,
  [ -2, -2 ], 1, [ 4, -2 ], 1, [ -4, 2 ], 1, [ 0, -6 ], 1 ]
gap> d:= DifferenceCharacter(ch1, ch2);
[ [ 4, 1 ], 1, [ 5, -1 ], 1, [ 3, 0 ], 1, [ 0, 3 ], 1, [ 1, 1 ], 1,
  [ 2, -1 ], 1, [ -1, 2 ], 1, [ -4, 5 ], 1, [ 3, -3 ], 1, [ 0, 0 ], 1,
  [ -3, 3 ], 1, [ 1, -2 ], 1, [ -2, 1 ], 1, [ -5, 4 ], 1, [ -1, -1 ], 1,
  [ 0, -3 ], 1, [ -3, 0 ], 1, [ 1, -5 ], 1, [ -1, -4 ], 1 ]
```

2.2 Decomposition numbers

If V is a given Weyl module, the command:

1 ► `DecompositionNumbers(` V `)` F

returns a list of highest weights of the composition factors of V, along with their corresponding multiplicities.

```
gap> V:= WeylModule(2, [8,0], "A", 2);
<Type A2 Weyl module of highest weight [ 8, 0 ] at prime p = 2>
gap> DecompositionNumbers(V);
[ [ 8, 0 ], 1, [ 6, 1 ], 1, [ 4, 2 ], 1, [ 0, 4 ], 1, [ 0, 1 ], 1 ]
gap> V:= WeylModule(3, [1,1], "A", 2);
<Type A2 Weyl module of highest weight [ 1, 1 ] at prime p = 3>
gap> DecompositionNumbers(V);
[ [ 1, 1 ], 1, [ 0, 0 ], 1 ]
```

2.3 Decomposing tensor products

One can also decompose tensor products of known modules, using the following functions.

1 ▶ `ProductCharacter( a, b )` F
▶ `DecomposeCharacter( ch, p, type, rank )` F

The function `ProductCharacter` returns the product of two given characters, and `DecomposeCharacter` computes the decomposition numbers of a given character *ch*, relative to simple characters in characteristic *p* for a given type and rank.

In the following example, we compute the multiplicities of simple composition factors in the tensor square of the natural representation for the group of Type A_4 in characteristic 2.

```
gap> ch:= SimpleCharacter(2, [1,0,0,0], "A", 4);
[ [ 1, 0, 0, 0 ], 1, [ -1, 1, 0, 0 ], 1, [ 0, -1, 1, 0 ], 1, [ 0, 0, -1, 1 ],
  1, [ 0, 0, 0, -1 ], 1 ]
gap> chsquared:= ProductCharacter(ch, ch);
[ [ 2, 0, 0, 0 ], 1, [ 0, 1, 0, 0 ], 2, [ 1, -1, 1, 0 ], 2, [ 1, 0, -1, 1 ],
  2, [ 1, 0, 0, -1 ], 2, [ -2, 2, 0, 0 ], 1, [ -1, 0, 1, 0 ], 2,
  [ -1, 1, -1, 1 ], 2, [ -1, 1, 0, -1 ], 2, [ 0, -2, 2, 0 ], 1,
  [ 0, -1, 0, 1 ], 2, [ 0, -1, 1, -1 ], 2, [ 0, 0, -2, 2 ], 1,
  [ 0, 0, -1, 0 ], 2, [ 0, 0, 0, -2 ], 1 ]
gap> DecomposeCharacter(chsquared, 2, "A", 4);
[ [ 2, 0, 0, 0 ], 1, [ 0, 1, 0, 0 ], 2 ]
```

3 Schur algebras and Symmetric Groups

In principle, the decomposition numbers for the algebraic group SL_n of Type A_{n-1} determine the decomposition numbers for Schur algebras, and thus determine also the decomposiotion numbers for symmetric groups. People working with Schur algebras and symmetric groups often prefer to use partitions to label highest weights. Of course, it is trivial to convert between SL_n weight and partition notation. This section covers functions that perform such conversions, and various other functions for Schur algebras and symmetric groups.

3.1 Compositions and Weights

A **composition** of degree r is a finite sequence $c = [c_1, \ldots, c_n]$ of non-negative integers which sum to r. The number n of parts of c is called its **length**. One may identify the set of compositions of length n with the set of **polynomial** weights of the algebraic group GL_n.

Note that a composition is a partition if and only if it is a dominant weight relative to the diagonal maximal torus in GL_n.

1 ▶ `CompositionToWeight( c )` F
▶ `WeightToComposition( r, wt )` F

`CompositionToWeight` converts a given list c (of length n) into an SL_n weight, by taking successive differences in the parts of c. This produces a list of length $n-1$.

`WeightToComposition` does the reverse operation, padding with zeros if necessary in order to return a composition of degree r. The degree must be specified since it is not uniquely determined by the given weight. Note that degree is unique modulo n. The length n of the output is always one more than the length of the input.

As a special case, these operations take partitions to dominant weights, and vice versa.

```
gap> wt:= CompositionToWeight( [3,3,2,1,1,0,0] );
[ 0, 1, 1, 0, 1, 0 ]
gap> WeightToComposition(10, wt);
[ 3, 3, 2, 1, 1, 0, 0 ]
gap> WeightToComposition(17, wt);
[ 4, 4, 3, 2, 2, 1, 1 ]
```

2 ▶ `BoundedPartitions( n, r, s )` F
▶ `BoundedPartitions( n, r )` F

`BoundedPartitions(`n`,`r`,`s`)` returns a list of n-part partitions of degree r, where each part lies in the interval $[0,s]$. Note that some parts of the partition may be equal to zero.

`BoundedPartitions(`n`,`r`)` is equivalent to `BoundedPartitions(`n`,`r`,`r`)`, which returns a list of all n-part partitions of degree r.

```
gap> BoundedPartitions(4,3,2);
[ [ 2, 1, 0, 0 ], [ 1, 1, 1, 0 ] ]
gap> BoundedPartitions(4,3,3);
[ [ 3, 0, 0, 0 ], [ 2, 1, 0, 0 ], [ 1, 1, 1, 0 ] ]
gap> BoundedPartitions(4,3);
[ [ 3, 0, 0, 0 ], [ 2, 1, 0, 0 ], [ 1, 1, 1, 0 ] ]
gap> BoundedPartitions(4,4,4);
[ [ 4, 0, 0, 0 ], [ 3, 1, 0, 0 ], [ 2, 2, 0, 0 ], [ 2, 1, 1, 0 ],
  [ 1, 1, 1, 1 ] ]
```

Running `BoundedPartitions(`n`,`n`,`n`)` produces a list of all partitions of n.

3.2 Schur Algebras

1 ▶ `SchurAlgebraWeylModule(` p, wt `)` F

`SchurAlgebraWeylModule` returns the Weyl module of highest weight wt in characteristic p, regarded as a module for GL_n where n is the length of the given partition wt.

```
gap> V:= SchurAlgebraWeylModule(3, [2,2,1,1,0]);
<Schur algebra Weyl module of highest weight [ 2, 2, 1, 1, 0 ] at prime p = 3>
```

2 ▶ `DecompositionNumbers(` V `)` F

`DecompositionNumbers` returns the decomposition numbers of a given Schur algebra Weyl module V, using partition notation for dominant weights.

```
gap> V:= SchurAlgebraWeylModule(3, [2,2,1,1,0]);
<Schur algebra Weyl module of highest weight [ 2, 2, 1, 1, 0 ] at prime p = 3>
gap> DecompositionNumbers(V);
[ [ 2, 2, 1, 1, 0 ], 1 ]
```

The above example shows an irreducible Weyl module for GL_5. Here is a more interesting example:

```
gap> V:= SchurAlgebraWeylModule(2, [4,2,1,1,0]);
<Schur algebra Weyl module of highest weight [ 4, 2, 1, 1, 0 ] at prime p = 2>
gap> DecompositionNumbers(V);
[ [ 4, 2, 1, 1, 0 ], 1, [ 3, 3, 1, 1, 0 ], 1, [ 3, 2, 2, 1, 0 ], 1,
  [ 4, 1, 1, 1, 1 ], 1, [ 2, 2, 2, 2, 0 ], 1, [ 2, 2, 2, 1, 1 ], 1 ]
```

3 ▶ `SchurAlgebraDecompositionMatrix(` p, n, r `)` F

`SchurAlgebraDecompositionMatrix` returns the decomposition matrix for a Schur algebra $S(n,r)$ in characteristic p. The rows and columns of the matrix are indexed by the partitions produced by `BoundedPartitions(`n`,`r`)`.

```
gap> SchurAlgebraDecompositionMatrix(2, 4, 5);
[ [ 1, 0, 1, 1, 0, 0 ], [ 0, 1, 0, 0, 0, 1 ], [ 0, 0, 1, 1, 1, 0 ],
  [ 0, 0, 0, 1, 1, 0 ], [ 0, 0, 0, 0, 1, 0 ], [ 0, 0, 0, 0, 0, 1 ] ]
gap> BoundedPartitions(4,5);
[ [ 5, 0, 0, 0 ], [ 4, 1, 0, 0 ], [ 3, 2, 0, 0 ], [ 3, 1, 1, 0 ],
  [ 2, 2, 1, 0 ], [ 2, 1, 1, 1 ] ]
```

3.3 Symmetric groups

Symmetric group decomposition numbers in positive characteristic may be obtained from corresponding decomposition numbers for a Schur algebra Weyl module, by means of the well known *Schur functor*. (See for instance Chapter 6 of [Gre07] for details.)

This is not a very efficient method to calculate those decomposition numbers. People needing such numbers for large partitions should use other methods. The approach taken here, through Schur algebras, is reasonably fast only up to about degree 7 at present.

1 ▶ `SymmetricGroupDecompositionNumbers( p, mu )` F

`SymmetricGroupDecompositionNumbers` returns a list of the decomposition numbers $[S_\mu : D_\lambda]$ for the dual Specht module S_μ labeled by a partition μ, in characteristic p. The simple modules D_λ are labeled by p-restricted partitions of the same degree as μ.

```
gap> SymmetricGroupDecompositionNumbers(3, [3,2,1]);
[ [ 3, 2, 1 ], 1, [ 2, 2, 2 ], 1, [ 3, 1, 1, 1 ], 1, [ 2, 1, 1, 1, 1 ], 1,
  [ 1, 1, 1, 1, 1, 1 ], 1 ]
```

2 ▶ `SymmetricGroupDecompositionMatrix( p, n )` F

`SymmetricGroupDecompositionMatrix` returns the decomposition matrix for the symmetric group on n letters, in characteristic p. The rows of the matrix are labeled by partitions of n and columns are labeled by p-restricted partitions of n. To obtain lists of the row and column labels use the functions

3 ▶ `AllPartitions( n )` F
▶ `pRestrictedPartitions( p, n )` F

which respectively return a list of all partitions of n, and a list of all p-restricted partitions of n. Note that trailing zeros are omitted in such partitions. (GAP has a built-in `Partitions` function that also gives all the partitions of n, but the ordering is different from the ordering in `AllPartitions`. To correctly interpret row labels in the decomposition matrix, one must use the ordering in `AllPartitions`.)

For example, we compute the decomposition matrix for the symmetric group on 5 letters in characteristic 2, along with the row and column labels for the matrix:

```
gap> SymmetricGroupDecompositionMatrix(2, 5);
[ [ 0, 0, 1 ], [ 0, 1, 0 ], [ 1, 0, 1 ], [ 1, 0, 2 ], [ 1, 0, 1 ],
  [ 0, 1, 0 ], [ 0, 0, 1 ] ]
gap> AllPartitions(5);
[ [ 5 ], [ 4, 1 ], [ 3, 2 ], [ 3, 1, 1 ], [ 2, 2, 1 ], [ 2, 1, 1, 1 ],
  [ 1, 1, 1, 1, 1 ] ]
gap> pRestrictedPartitions(2,5);
[ [ 2, 2, 1 ], [ 2, 1, 1, 1 ], [ 1, 1, 1, 1, 1 ] ]
```

3.4 The Mullineux correspondence

Computing symmetric group decomposition numbers by means of the Schur functor naturally produces the numbers $[S_\mu : D_\lambda]$ for partitions μ and p-restricted partitions λ. Here S^μ is the dual Specht module labeled by μ and D_λ the dual simple module labeled by μ.

Let λ' be the conjugate partition of a partition λ, obtained by transposing rows and columns of the corresponding Young diagram. We have $(S^\mu)^* \simeq S_\mu$ for a partition μ and $D^\lambda \otimes \mathrm{sgn} \simeq D_{\lambda'}$ for a p-regular partition λ, where S^μ is the usual Specht module and D^λ the usual simple module, using notation in accord with [Jam78]. The notation sgn refers to the sign representation.

Thus it follows that $[S_\mu : D_\lambda] = [S^\mu : D^{\mathrm{Mull}(\lambda')}]$ if λ is p-restricted. So by sending $\lambda \to \mathrm{Mull}(\lambda')$, one obtains the column labels for the decomposition matrix that appear in [Jam78].

1 ▶ `Mullineux( p, mu )` F

`Mullineux` returns the partition $\mathrm{Mull}(\mu)$ corresponding to a given p-regular partition μ under the Mullineux map. This means by definition that $D^\mu \otimes \mathrm{sgn} \simeq D^{\mathrm{Mull}(\mu)}$.

```
gap> Mullineux(3, [5,4,1,1]);
[ 9, 2 ]
gap> Mullineux(3, [9,2]);
[ 5, 4, 1, 1 ]
```

2 ▶ `pRegularPartitions( p, n )` F

`pRegularPartitions` returns a list of the p-regular partitions of n, in bijection with the list of p-restricted partitions of n produced by `pRestrictedPartitions`, using the bijection $\lambda \to \mathrm{Mull}(\lambda')$. Thus, to read a symmetric group decomposition matrix using p-regular partition notation, one uses the output of `pRegularPartitions` to index the columns of the matrix.

```
gap> SymmetricGroupDecompositionMatrix(3, 5);
[ [ 0, 0, 1, 0, 0 ], [ 1, 0, 0, 0, 0 ], [ 1, 0, 0, 0, 1 ], [ 0, 1, 0, 0, 0 ],
  [ 0, 0, 1, 1, 0 ], [ 0, 0, 0, 1, 0 ], [ 0, 0, 0, 0, 1 ] ]
gap> AllPartitions(5);
[ [ 5 ], [ 4, 1 ], [ 3, 2 ], [ 3, 1, 1 ], [ 2, 2, 1 ], [ 2, 1, 1, 1 ],
  [ 1, 1, 1, 1, 1 ] ]
gap> pRegularPartitions(3, 5);
[ [ 4, 1 ], [ 3, 1, 1 ], [ 5 ], [ 2, 2, 1 ], [ 3, 2 ] ]
```

In the above example, we computed a decomposition matrix in characteristic 3 for the symmetric group on 5 letters, along with labels for its rows and columns, using p-regular partitions to label the columns. If one wants instead to use p-restricted column labels, one needs to run:

```
gap> pRestrictedPartitions(3, 5);
[ [ 3, 2 ], [ 3, 1, 1 ], [ 2, 2, 1 ], [ 2, 1, 1, 1 ], [ 1, 1, 1, 1, 1 ] ]
```

from the preceding section, to see the correspondence.

3.5 Miscellaneous functions for partitions

Here are a few additional miscellaneous functions useful for computing with partitions.

1 ▶ `Conjugate( mu )` F

`Conjugate` returns the conjugate partition μ' of its input μ. The Young diagram of μ' is obtained from that of μ by transposing rows and columns. For example,

```
gap> Conjugate( [4,4,2,1] );
[ 4, 3, 2, 2 ]
gap> Conjugate( [4,3,2,2] );
[ 4, 4, 2, 1 ]
```

2 ▶ `pRestricted( p, mu )` F
▶ `pRegular( p, mu )` F

`pRestricted` returns true iff the partition *mu* is p-restricted (succesive differences $\mu_i - \mu_{i+1}$ are strictly bounded above by p); similarly `pRegular` returns true iff the partition *mu* is p-regular (equivalently, the conjugate of *mu* is p-restricted).

```
gap> pRestricted(3, [6,6,6,4,4,2,1,1]);
true
gap> pRestricted(3, [6,3,1,1]);
false
gap> pRegular(3, [8,6,5,5,3,3]);
true
gap> pRegular(3, [3,3,3,2,2,1,1]);
false
```

Bibliography

[CPS75] Edward Cline, Brian Parshall, and Leonard Scott, Cohomology of finite groups of Lie type, I, *Inst. Hautes Études Sci. Publ. Math.* 45 (1975), 169–191.

[DM08] Stephen Doty and Stuart Martin, Decomposition of tensor products of modular irreducible representations for SL_3, *preprint* 2008.

[Irv86] Ronald S. Irving, The structure of certain highest weight modules for SL_3, *J. Algebra* 99 (1986), 438–457.

[Gre07] J.A. Green, *Polynomial representations of* GL_n, Second corrected and augmented edition; With an appendix on Schensted correspondence and Littelmann paths by K. Erdmann, Green and M. Schocker; Lecture Notes in Mathematics, 830, Springer, Berlin, 2007.

[Jam78] Gordon D. James, *The representation theory of the symmetric groups*, Lecture Notes in Mathematics, 682, Springer, Berlin, 1978.

[Jan03] Jens C. Jantzen, *Representations of Algebraic Groups, Second edition*, Mathematical Surveys and Monographs, 107, American Mathematical Society, Providence, RI, 2003.

[Xi99] Nanhua Xi, Maximal and primitive elements in Weyl modules for type A_2. *J. Algebra* 215 (1999), 735–756.

Index

This index covers only this manual. A page number in *italics* refers to a whole section which is devoted to the indexed subject. Keywords are sorted with case and spaces ignored, e.g., "`PermutationCharacter`" comes before "permutation group".