WeylModules

for simple simply-connected algebraic groups

Version 2.0

29 February 2024

Stephen Doty

Stephen Doty
Email: doty@math.luc.edu
Homepage: https://doty.math.luc.edu
Address: Department of Mathematics and Statistics
Loyola University Chicago
Chicago, Illinois 60660 USA

Abstract

WeylModules is a GAP Package supporting computer computations with Weyl modules for simple simply-connected algebraic groups.

Copyright

Acknowledgements

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.

In 2018, Chris Bendel, Dan Nakano, Cornelius Pillen, and Paul Sobaje found the first counterexample to Donkin's tilting module conjecture using this package. Shortly thereafter, the author discovered an error in one of the functions. This spurred a major rewrite in the summer of 2019, resulting in Version 2.0 of the package. The rewriting of the documentation was delayed by the Covid19 Pandemic and inertia, and finally completed in February 2024.

Contents

Chapter 1

Introduction

This GAP Package supports digital computer computations with Weyl modules for a given simple 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 relatively small relative to the Coxeter number. The general theory of Weyl modules [Jan03] includes a number of basic properties that break down (or are not known to hold) if the characteristic is too small. In such cases, explicit computations are often useful.

Recall that a *maximal vector* is a weight vector which is killed by the positive unipotent radical; equivalently, it is killed by the positive part of the algebra of distributions.

The main technical idea underlying this package is the following fact: computing all the maximal vectors in a given Weyl module V classifies the nonzero Weyl modules W for which a nonzero homomorphism from W into V exists. Such homological information is a powerful aid to understanding structural properties of the Weyl module V. The implementation of this idea involves a brute force search through each dominant weight space, examing all linear combinations (over the prime field) and compiling a list of the ones which are maximal. This exploits the pleasant fact that for Weyl modules of small dimension, the weight spaces tend to be small enough to be manageable.

Versions 1.0 and 1.1 of this package were released in 2009. Their initial development was spurred by work on the paper [BDM11]. The discovery of the first counterexample to Donkin's tilting module conjecture, by Bendel, Nakano, Pillen, and Sobaje [BNPS20] motivated the author to make subsequent calculations that led to the realization that the `SubmoduleStructure` function was not reliable. This led to the development of Version 2.0, which eliminates the unreliable command and adds a number of new functions that extend the capabilities of the previous version.

Although most of the functions deal with the simple simply-connected case, there are a few functions for dealing with Schur algebras and symmetric groups. Those commands are limited in scope, and provided mainly as a convenience.

Chapter 2

Weyl modules

This chapter discusses the commands available for computations (in positive characteristic p) with Weyl modules, quotient Weyl modules, submodules of Weyl modules, and submodules of quotient Weyl modules.

WARNING. In most cases, the dimension of space of maximal vectors of a given dominant weight is either 0 or 1. Cases for which there exist two or more independent maximal vectors of the same weight can lead to complications, such as a lack of rigidity in the submodule structure. 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 on page 173 of [CPS75]. (I am grateful to Anton Cox for this reference.)

A Weyl module (as in the previous paragraph) with at least one weight space containing multiple (linearly independent) maximal vectors is called *ambiguous*. Extra care is needed when studying ambiguous Weyl modules.

2.1 Creating Weyl modules

There are two functions for creating a Weyl module.

```
WeylModule( p, wt, type, rank )
WeylModule( V, wt )
```

In both forms *wt* is the highest weight of the Weyl module. The function with four arguments specifies the characteristic *p*, the root system *type* (a letter in the range A–G) and its *rank*. In the second form, with two arguments, the function gets the characteristic and underlying root system from an existing Weyl module *V*.

Example

```
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.

```
IsWeylModule( V )
```

This returns true if *V* is a Weyl module and returns false otherwise.

Example

```
gap> IsWeylModule(W);
true
gap> IsWeylModule(4);
false
```

2.2 Creating quotient Weyl modules

Quotients of an existing Weyl module by a submodule are supported. See Section 2.3 below for methods that create submodules.

```
QuotientWeylModule( S )
DefiningKernel( Q )
```

The first function returns the quotient module *V/S*, where *V* is the ambient Weyl module of which *S* is a submodule. In the latter function, *Q* must be an existing quotient Weyl module and its kernel (a submodule) is returned.

Example

```
gap> Q := QuotientWeylModule(D);
<Quotient of Type G2 Weyl module of highest weight [ 2, 0 ] at prime p = 2>
```

In the above, it is assumed that *D* is the submodule constructed in the second example of Section 2.3 below.

There is also a category of quotient Weyl modules.

```
gap> IsQuotientWeylModule( Q )
```

which returns true if *Q* is a quotient Weyl module, and returns false otherwise.

2.3 Creating submodules

The WeylModules package includes support for working with submodules. Submodules of ambient modules are created by specifying either a single generator or a list of generators; in the latter case the submodule is the linear sum of the submodules generated by the vectors on the list. The ambient module can be: a Weyl module, a quotient of a Weyl module, or an existing submodule of either a Weyl module or a quotient Weyl module.

```
SubWeylModule( V, vec )
SubWeylModule( V, lst )
```

This returns a submodule of the given ambient object *V* of one of the forms described above. If the second argument *vec* is a vector, then it returns the submodule of *V* generated by that vector. If the second argument is a list *lst* then it returns the submodule generated by that list of vectors.

WARNING: If the dimension of the ambient module is large, this can take a very long time.

Example

```
gap> V := WeylModule(3,[3,3],"A",2);
<Type A2 Weyl module of highest weight [ 3, 3 ] at prime p = 3>
gap> m := MaximalVectors(V);
[ 1*v0, y1*v0, y2*v0, y1^(2)*y2*v0, -1*y1*y2^(2)*v0+y2*y3*v0,
  y1*y2*y3*v0+y1^(2)*y2^(2)*v0 ]
gap> List(m,Weight);
[ [ 3, 3 ], [ 1, 4 ], [ 4, 1 ], [ 0, 3 ], [ 3, 0 ], [ 1, 1 ] ]
gap> S := SubWeylModule(V, m[6]);
<SubWeylModule of dimension 7, generated by elements
[ y1*y2*y3*v0+y1^(2)*y2^(2)*v0 ] of weights [ [ 1, 1 ] ]>, in
<Type A2 Weyl module of highest weight [ 3, 3 ] at prime p = 3>
gap> S2 := SubWeylModule(V, [m[4],m[5]]);
<SubWeylModule of dimension 13, generated by elements
[ y1^(2)*y2*v0, -1*y1*y2^(2)*v0+y2*y3*v0 ] of weights [ [ 0, 3 ], [ 3, 0 ]
 ]>, in
<Type A2 Weyl module of highest weight [ 3, 3 ] at prime p = 3>
```

An existing submodule remembers its list of generators.

`Generators( S )`

returns the list of generators used in creating the submodule *S*. Here, *S* must be either a sub Weyl module or a subquotient.

Example

```
gap> Generators(S2);
[ y1^(2)*y2*v0, -1*y1*y2^(2)*v0+y2*y3*v0 ]
```

In the above example, it is assumed that S2 is the submodule constructed in the preceding example.

There is another function that creates submodules of an existing Weyl module or quotient Weyl module, as a direct sum of a list of existing submodules. The submodules in the list must be linearly independent.

`SubWeylModuleDirectSum( V, lst )`

This returns the direct sum of the given list of submodules, as a submodule of the ambient module *V*.

Example

```
gap> V := WeylModule(2,[2,0],"G",2);
<Type G2 Weyl module of highest weight [ 2, 0 ] at prime p = 2>
gap> m := MaximalVectors(V);
[ 1*v0, y1*v0, y4*v0 ]
gap> List(m,Weight);
[ [ 2, 0 ], [ 0, 1 ], [ 1, 0 ] ]
gap> S1 := SubWeylModule(V,m[2]);
<SubWeylModule of dimension 14, generated by elements [ y1*v0 ] of weights
[ [ 0, 1 ] ]>, in
<Type G2 Weyl module of highest weight [ 2, 0 ] at prime p = 2>
gap> S2 := SubWeylModule(V,m[3]);
<SubWeylModule of dimension 6, generated by elements [ y4*v0 ] of weights
[ [ 1, 0 ] ]>, in
```

```
<Type G2 Weyl module of highest weight [ 2, 0 ] at prime p = 2>
gap> D := SubWeylModuleDirectSum(V,[S1,S2]);
<SubWeylModule of dimension 20, generated by elements
[ y1*v0, y4*v0 ] of weights [ [ 0, 1 ], [ 1, 0 ] ]>, in
<Type G2 Weyl module of highest weight [ 2, 0 ] at prime p = 2>
```

In the above example, the given Weyl module of type G_2 in characteristic 2 has two maximal vectors of different weights in the socle, so it is possible to form the direct sum of the submodules thay generate. In this case, the direct sum coincides with the socle. (The `SocleWeyl` command calls the above to produce the socle directly.)

We also have categories of submodules.

```
IsSubWeylModule ( X )
IsSubQuotientWeylModule( X )
```

These functions return true if the object is a submodule of an existing Weyl module or existing quotient Weyl module, respectively, and return false otherwise.

Example

```
gap> IsSubWeylModule(D);
true
gap> IsSubQuotientWeylModule(D);
false
```

Here, we assume that D is as defined in the preceding example.

2.4 Weights and weight spaces

The following functions are available for computing weights and weight spaces in a given Weyl module, quotient, submodule, or subquotient.

```
Weight( vec )
Weights( V )
DominantWeights( V )
WeightSpaces( V )
DominantWeightSpaces( V )
WeightSpace( V, wt )
```

The function `Weight` returns the weight of the given weight vector *vec*. `Weights` returns a list of all the weights of *V*, without multiplicities. `DominantWeights` returns a list of all the dominant weights of *V*, again without multiplicities. `WeightSpaces` returns a list consisting of each weight followed by a basis of weight vectors for the corresponding weight space in *V*, and `WeightSpace` returns a basis for the weight space of the given weight *wt*.

In all of these functions, *V* can be an existing Weyl module, quotient Weyl module, submodule, or subquotient.

Example

```
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.

2.4.1 Using `List` with `Weight`

The builtin `List` function in GAP supports mapping with respect to another function as an optional second argument. If *`lst`* is a given list of weight vectors (for instance a basis of a module) then the function

```
List( lst, Weight )
```

returns a list of the weights of the vectors in *`lst`*. In other words, it applies `Weight` to each element in *`lst`*. This can be very handy.

Example

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

In the above, we compute a basis of weight vectors and the corresponding list of weights for the natural module of type A_2.

2.5 Maximal vectors

As mentioned at the beginning of this chapter, the commands to compute maximal vectors are fundamental for the WeylModules package. The command has two basic forms. In both forms, *`V`* is a given Weyl module, quotient Weyl module, or sub quotient Weyl module. In the second form, *`wt`* is a specified (dominant) weight.

```
MaximalVectors( V )
MaximalVectors( V, lst )
```

This command returns a list of maximal vectors in the specified weight space, or, if no weight is specified, a list of all the maximal vectors in the module.

Example

```
gap> V := WeylModule(3,[3,3],"A",2);
<Type A2 Weyl module of highest weight [ 3, 3 ] at prime p = 3>
gap> m := MaximalVectors(V);
[ 1*v0, y1*v0, y2*v0, y1^(2)*y2*v0, -1*y1*y2^(2)*v0+y2*y3*v0,
  y1*y2*y3*v0+y1^(2)*y2^(2)*v0 ]
gap> List(m,Weight);
[ [ 3, 3 ], [ 1, 4 ], [ 4, 1 ], [ 0, 3 ], [ 3, 0 ], [ 1, 1 ] ]
m11 := MaximalVectors(V,[1,1]);
[ y1*y2*y3*v0+y1^(2)*y2^(2)*v0 ]
```

There is also a function for testing a given vector in a Weyl module or a quotient Weyl module, to see if it is maximal or not.

`IsMaximalVector( V, vec )`

This returns true of the given *vec* is maximal in *V*, which can be a Weyl module or a quotient Weyl module, and returns false otherwise.

Example

```
gap> V;
<Type A2 Weyl module of highest weight [ 3, 3 ] at prime p = 3>
gap> m[5];
-1*y1*y2^(2)*v0+y2*y3*v0
gap> IsMaximalVector(V,m[5]);
true
```

The above assumes that *V* is the Weyl module defined in the previous example, and that *m* is the list of its maximal vectors.

2.6 Ambiguity

Recall from the beginning of this chapter that we defined an *ambiguous* Weyl module to be one which has a weight space containing at least two linearly independent maximal vectors. Extra care is needed when computing with such modules, so the `MaximalVectors` function produces a warning message and tags the module as `Ambiguous`. Here is an example, which was found in [CPS75].

Example

```
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: Ambiguous module detected *****
[ 1*v0, y5*y10*v0+y6*y9*v0, y2*y11*v0+y5*y10*v0+y12*v0 ]
```

In such situations, the following commands can be used.

`IsAmbiguous( V )`
`AmbiguousMaxVecs( V )`

The first function, which returns true or false, checks to see if the given module *V* is ambiguous. The second function returns a list of lists of the ambiguous maximal vectors in the various ambiguous weight spaces. In both functions, *V* is a Weyl module, quotient, submodule. (WHY NOT subquotient??)

Example

```
gap> IsAmbiguous(V);
true
gap> m := AmbiguousMaxVecs(V);
[ [ y5*y10*v0+y6*y9*v0, y2*y11*v0+y5*y10*v0+y12*v0 ] ]
gap> List(m[1],Weight);
[ [ 0, 0, 0, 0 ], [ 0, 0, 0, 0 ] ]
```

Here, we see that V as above has two maximal vectors of weight zero. This means that the space of homomorphisms from the trivial module into V is two-dimensional.

2.7 Basis and dimension

One can compute the dimension or a basis of weight vectors for a given module.

```
Dim( V )
BasisVecs( V )
```

This returns the dimension and a list of weight vectors, resprectively. The module `V` must be a Weyl module, a submodule, a quotient, or a subquotient.

Example

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

2.8 Miscellaneous commands

There are a few additional commands available for a Weyl module or a quotient Weyl module.

```
Generator( V )
TheLieAlgebra( V )
SimpleLieAlgGens( V )
TheCharacteristic( V )
```

These commands respectively return the generator, the underlying Lie algebra, a list of the Lie algebra generators, and the characteristic. The third function is only available for a Weyl module (it is used internally by some of the other functions). The Lie algebra is the *characteristic zero* Lie algebra of the same type as the underlying algebraic group. The module `V` must be either a Weyl module or a quotient Weyl module.

Example

```
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> g:= Generator(V);
```

```
1*v0
gap> TheCharacteristic(V);
2
```

Chapter 3

Characters and structure

Given a Weyl module, one can compute certain structural information such as its socle series or its unique simple quotient. One can also compute characters of Weyl modules, quotients, submodules, and subquotients. Characters of tensor products are also supported, in a limited way.

In particular, it is possible to compute all the simple characters for a given root system and characteristic, provided the modules do not get too large. And one can compute the decomposition numbers for a given Weyl module, again assuming it is not too large.

3.1 Characters

The most basic command to compute the character of an existing module is the following.

```
Character( V )
```

This returns a list consisting of each weight of *V* followed by its corresponding weight space dimension. The module *V* must be a Weyl module, a submodule, a quotient, or a subquotient.

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);
<SubWeylModule of dimension 7, generated by elements [ y1*v0 ] of weights
[ [ 1, 1 ] ]>, in
<Type A2 Weyl module of highest weight [ 3, 0 ] at prime p = 3>
gap> Character(S);
[ [ 1, 1 ], 1, [ -1, 2 ], 1, [ 2, -1 ], 1, [ 0, 0 ], 1, [ -2, 1 ], 1,
  [ 1, -2 ], 1, [ -1, -1 ], 1 ]
gap> Character(QuotientWeylModule(S));
[ [ 3, 0 ], 1, [ -3, 3 ], 1, [ 0, -3 ], 1 ]
```

Simple characters for a given highest (dominant) weight are computed by one of the following.

```
SimpleCharacter( p, wt, type, rk )
SimpleCharacter( V, wt )
```

These return the simple character of the given highest weight. In the second form, *V* must be an existing Weyl module, and the characteristic *p*, the type *type*, and the rank *rk* are looked up in *V*. (Note: In the second form, the highest weight *wt* does not have to agree with the highest weight of *V*.)

Example

```
gap> char:= SimpleCharacter(3, [3,0], "A", 2);
[ [ 3, 0 ], 1, [ -3, 3 ], 1, [ 0, -3 ], 1 ]
gap> V:= WeylModule(2, [1,0], "G", 2);
<Type G2 Weyl module of highest weight [ 1, 0 ] at prime p = 2>
gap> char := SimpleCharacter(V,[0,2]);
[ [ 0, 2 ], 1, [ 6, -2 ], 1, [ 2, 0 ], 1, [ -2, 2 ], 1, [ -6, 4 ], 1,
  [ 4, -2 ], 1, [ 0, 0 ], 2, [ 6, -4 ], 1, [ -4, 2 ], 1, [ 2, -2 ], 1,
  [ -2, 0 ], 1, [ -6, 2 ], 1, [ 0, -2 ], 1 ]
```

Characters can be multiplied. Given two characters (of modules) for the same root system, it is possible to compute the character of the tensor product of the modules. This is used internally in implementing Steinberg's tensor product theorem.

`ProductCharacter( ch1, ch2 )`

Returns the product character.

Example

```
gap> V := WeylModule(3,[1,1],"A",2);
<Type A2 Weyl module of highest weight [ 1, 1 ] at prime p = 3>
gap> ch1 := SimpleCharacter(V,[0,1]);
[ [ 0, 1 ], 1, [ 1, -1 ], 1, [ -1, 0 ], 1 ]
gap> prod := ProductCharacter(ch1,ch1);
[ [ 0, 2 ], 1, [ 1, 0 ], 2, [ -1, 1 ], 2, [ 2, -2 ], 1, [ 0, -1 ], 2,
  [ -2, 0 ], 1 ]
```

It is possible to subtract characters. The result is not in general a character. The function `DecomposeCharacter` uses character subtraction to decompose a given character into its simple characters with multiplicity. This is used internally by the `DecompositionNumbers` function to compute the decomposition numbers of a Weyl module.

`DifferenceCharacter( ch1, ch2 )`
`DecomposeCharater( ch, p, type, rk )`

The first function returns the virtual character of the difference of its inputs. The second computes the decomposition numbers of the given charaacter *ch* for the root system of type *type*, rank *rk*, and characteristic *p*.

Example

```
gap> d := DecomposeCharacter(prod,3,"A",2);
[ [ 0, 2 ], 1, [ 1, 0 ], 1 ]
gap> V := WeylModule(3,[1,1],"A",2);
<Type A2 Weyl module of highest weight [ 1, 1 ] at prime p = 3>
gap> dn := DecomposeCharacter(Character(V),3,"A",2);
[ [ 1, 1 ], 1, [ 0, 0 ], 1 ]
```

In the first example, `prod` is the product character computed in the previous example; thus we have decomposed a tensor product. In the last example, `dn` gives the decomposition numbers of the Weyl module of highest weight [1,1] in characteristic 3. (This can also be computed more directly by the `DecompositionNumbers` function.)

3.2 Decomposition numbers

The decomposition numbers of a given Weyl module can be computed.

`DecompositionNumbers( V )`

This returns a list consisting of each highest weight of a simple composition factor followed by its composition factor multiplicity. It is assumed that *V* is an existing Weyl module. (There are other ways to compute this, so the function is provided as a convenience for the user.)

Example

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

3.3 The maximal submodule and simple quotient

A given Weyl module *V* always has a unique maximal submodule. The package supports computing this important submodule.

`MaximalSubmodule( V )`

This returns the submodule generated by all proper submodules.

Example

```
gap> V := WeylModule(3,[3,0],"A",2);
<Type A2 Weyl module of highest weight [ 3, 0 ] at prime p = 3>
gap> S := MaximalSubmodule(V);
<SubWeylModule of dimension 7, generated by elements [ y1*v0 ] of weights
[ [ 1, 1 ] ]>, in
<Type A2 Weyl module of highest weight [ 3, 0 ] at prime p = 3>
```

Here, the Weyl module is the third symmetric power of the natural module. It has dimension 10. The maximal submodule makes a list of its generators available to the `Generators` function for further computation.

Let's check that the difference character of the Weyl character and its maximal submodule coincides with the corresponding simple character.

Example

```
gap> d := DifferenceCharacter(Character(V),Character(S));
[ [ 3, 0 ], 1, [ -3, 3 ], 1, [ 0, -3 ], 1 ]
gap> s := SimpleCharacter(3,[3,0],"A",2);
[ [ 3, 0 ], 1, [ -3, 3 ], 1, [ 0, -3 ], 1 ]
```

We can also compute the corresponding simple quotient of *V* by its maximal submodule, as a quotient Weyl module. The general form is

`SimpleQuotient( V )`

This returns the simple quotient of a Weyl module *V* by its unique maximal submodule.

Example

```
gap> V := WeylModule(3,[3,0],"A",2);
<Type A2 Weyl module of highest weight [ 3, 0 ] at prime p = 3>
gap> Q := SimpleQuotient(V);
<Quotient of Type A2 Weyl module of highest weight [ 3, 0 ] at prime p = 3>
gap> ch := Character(Q);
[ [ 3, 0 ], 1, [ -3, 3 ], 1, [ 0, -3 ], 1 ]
```

Notice that this gives yet another way to calculate simple characters.

3.4 The socle series

The package supports computing the socle series of a given Weyl module, or of a given quotient Weyl module.

```
SocleSeries( V )
```

This returns the socle series, as a list of submodules (or a list of subquotients, if *V* is a quotient Weyl module).

Example

```
gap> V := WeylModule(3,[3,0],"A",2);
<Type A2 Weyl module of highest weight [ 3, 0 ] at prime p = 3>
gap> SocleSeries(V);
[ <SubWeylModule of dimension 7, generated by elements [ y1*v0 ] of weights
    [ [ 1, 1 ] ]>, in
    <Type A2 Weyl module of highest weight [ 3, 0 ] at prime p = 3>,
  <SubWeylModule of dimension 10, generated by elements
    [ y1*v0, 1*v0 ] of weights [ [ 1, 1 ], [ 3, 0 ] ]>, in
    <Type A2 Weyl module of highest weight [ 3, 0 ] at prime p = 3> ]
```

The socle series is computed by internally applying the following functions, which may be useful in other situations as well.

```
SocleWeyl( V )
NextSocle( S )
GensNextSocle( S )
```

The first function returns the socle of its argument. The second function returns the next socle layer, assuming that *S* is a submodule. That is, `NextSocle` returns the submodule that maps onto the socle of the quotient *V*/*S*, where *V* is the ambient module. The third function returns a list of generators of the next socle layer for the given submodule *S*. For the first two functions, *V* must be either a Weyl module or a quotient Weyl module. The third function is only available for a Weyl module.

Example

```
gap> soc := SocleWeyl(V);
<SubWeylModule of dimension 7, generated by elements
[ y1*y2*y3*v0+y1^(2)*y2^(2)*v0 ] of weights [ [ 1, 1 ] ]>, in
<Type A2 Weyl module of highest weight [ 3, 3 ] at prime p = 3>
```

Example

```
gap> V := WeylModule(3,[3,3],"A",2);
<Type A2 Weyl module of highest weight [ 3, 3 ] at prime p = 3>
gap> m := MaximalVectors(V);
[ 1*v0, y1*v0, y2*v0, y1^(2)*y2*v0, -1*y1*y2^(2)*v0+y2*y3*v0,
  y1*y2*y3*v0+y1^(2)*y2^(2)*v0 ]
gap> S := SubWeylModule(V,[m[4],m[5]]);
<SubWeylModule of dimension 13, generated by elements
[ y1^(2)*y2*v0, -1*y1*y2^(2)*v0+y2*y3*v0 ] of weights [ [ 0, 3 ], [ 3, 0 ]
 ]>, in
<Type A2 Weyl module of highest weight [ 3, 3 ] at prime p = 3>
gap> N := NextSocle(S);
<SubWeylModule of dimension 14, generated by elements
[ y1^(2)*y2*v0, -1*y1*y2^(2)*v0+y2*y3*v0, y1*y2*y3^(2)*v0 ] of weights
[ [ 0, 3 ], [ 3, 0 ], [ 0, 0 ] ]>, in
<Type A2 Weyl module of highest weight [ 3, 3 ] at prime p = 3>
gap> gens := GensNextSocle(S);
[ y1*y2*y3^(2)*v0 ]
```

3.5 Testing membership

It is possible to test a given weight vector for membership in a submodule or a subquotient.

```
IsWithin( S, vec )
```

This returns true if the given vector *vec* is in the submodule *S*, and returns false otherwise. The vector *vec* must be in the ambient module. Here, *S* can be either a submodule of a Weyl module, or a submodule of a quotient Weyl module.

Example

```
gap> V := WeylModule(3,[3,3],"A",2);
<Type A2 Weyl module of highest weight [ 3, 3 ] at prime p = 3>
gap> M := MaximalSubmodule(V);
<SubWeylModule of dimension 57, generated by elements
[ y1*v0, y2*v0, y1^(3)*y2^(3)*v0+y3^(3)*v0 ] of weights
[ [ 1, 4 ], [ 4, 1 ], [ 0, 0 ] ]>, in
<Type A2 Weyl module of highest weight [ 3, 3 ] at prime p = 3>
gap> m := MaximalVectors(V);
[ 1*v0, y1*v0, y2*v0, y1^(2)*y2*v0, -1*y1*y2^(2)*v0+y2*y3*v0,
  y1*y2*y3*v0+y1^(2)*y2^(2)*v0 ]
gap> IsWithin(M,m[1]);
false
gap> IsWithin(M,m[2]);
true
gap> IsWithin(M, m[2]+2*m[3]);
true
```

3.6 The ambient module

Submodules and quotient modules remember the original ambient Weyl modules from which they were constructed. Subquotients, which are always constructed as submodules in some quotient Weyl

module, know the ambient quotient in which they were constructed. The following functions return the ambient module to make it available for further computations.

```
AmbientWeylModule( M )
AmbientQuotient( S )
```

In the first function, *M* can be either a quotient Weyl module or a submodule of a Weyl module, and the ambient Weyl module is returned. In the second function, *M* must be a submodule of some existing quotient Weyl module, and the ambient quotient is returned.

Example

```
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> S := SocleWeyl(V);
***** WARNING: Ambiguous module detected *****
<SubWeylModule of dimension 2, generated by elements
[ y5*y10*v0+y6*y9*v0, y2*y11*v0+y5*y10*v0+y12*v0 ] of weights
[ [ 0, 0, 0, 0 ], [ 0, 0, 0, 0 ] ]>, in
<Type D4 Weyl module of highest weight [ 0, 1, 0, 0 ] at prime p = 2>
gap> AmbientWeylModule(S);
<Type D4 Weyl module of highest weight [ 0, 1, 0, 0 ] at prime p = 2>
```

3.7 Acting on a vector

It is sometimes useful to act on a given vector by an element of the hyperalgebra (algebra of distributions). One needs to use a builtin GAP function to construct elements of the hyperalgebra.

```
ActOn( V, u, vec )
```

This returns the result, in the ambient module *V*, of letting *u* act on the given vector *vec*, where *V* is either a Weyl module or a quotient Weyl module.

Example

```
gap> V:= WeylModule(2, [1,0], "G", 2);
<Type G2 Weyl module of highest weight [ 1, 0 ] at prime p = 2>
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 `LatticeGeneratorsInUEA` is an existing GAP command.

Chapter 4

Schur algebras and symmetric groups

The decomposition numbers for the algebraic group SL_n of type A_{n-1} determine the decomposition numbers for the corresponding Schur algebras, and thus also determine the decomposition numbers for symmetric groups. People working with Schur algebras and symmetric groups often prefer to use *partitions* to label highest weights. Although it is trivial to convert between SL_n weight notation and partition notation, for the sake of convenience, we provide a few functions that perform such conversions, and various other functions related to Schur algebras and symmetric groups.

It should be noted that the functions for symmetric groups are quite slow, so readers interested in symmetric group computations may want to look elsewhere for better tools.

4.1 Compositions and weights

A (weak) *composition* of degree r is a finite sequence $c = [c_1, \ldots, c_n]$ of non-negative integers which add up to r. The number n of parts of c is called its *length*. Note that zeros are allowed in weak compositions, which are called compositions from now on.

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.

```
CompositionToWeight( c )
WeightToComposition( r , wt )
```

`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. (The 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.

Example

```
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 ]
```

4.2 Bounded partitions

There are two functions that compute lists of bounded partitiuons.

```
BoundedPartitions( n, r, s )
BoundedPartitions( n, r )
```

The first form returns a list of all partitions of `n` parts in degree `r` whose parts lie in the closed interval $[0,s]$. The second form is equivalent to the first where $s = r$; that is, it returns a list of all of `n` parts in degree `r`.

Example

```
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 ] ]
```

Notice that we sometimes allow zero parts in partitions.

4.3 Listing partitions

We can also compute a list of all partitions of a given degree.

```
AllPartitions( r )
```

This returns a list of all the partitions of `r`, where zero parts are not allowed.

Example

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

One can also calculate lists of p-restricted and p-regular partitions, where p is a given prime.

```
pRestrictedPartitions( p, n )
pRegularPartitions( p, n )
```

Here `p` is the characteristic and `n` the degree.

Example

```
gap> pRestrictedPartitions(2,5);
[ [ 2, 2, 1 ], [ 2, 1, 1, 1 ], [ 1, 1, 1, 1, 1 ] ]
gap> pRegularPartitions(2,5);
[ [ 3, 2 ], [ 4, 1 ], [ 5 ] ]
```

Finally, we have functions that test a given partition to check whether or not it is restricted or regular with respect to a given prime.

```
pRestricted( p, ptn )
pRegular( p, ptn )
```

The functions return true if the given partition `ptn` is p-restricted or p-regular, respectively, and return false otherwise. The characteristic is p.

Example

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

4.4 Schur algebras

There are only a few functions for working directly with Schur algebra Weyl modules.

```
SchurAlgebraWeylModule( p, lambda )
```

This returns a Weyl module of highest weight `lambda` in characteristic p, regarded as a module for GL_n where n is the length of the given partition `lambda`. Note that zero parts are allowed in `lambda`.

Example

```
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>
```

There is also a category of Schur algebra Weyl modules.

```
IsSchurAlgebraWeylModule( V )
```

This returns true if the given `V` belongs to the category, and returns false otherwise.

It is possible to compute the decomposition numbers of a Schur algebra Weyl module, and the result is returned using partition notation.

```
DecompositionNumbers( V )
```

This returns a list consiting of each highest weight of a composition factor (in partition notation) followed by its corresponding composition factor multiplicity.

Example

```
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 ]
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 ]
```

Finally, we can compute the entire decomposition matrix for a given Schur algebra.

`SchurAlgebraDecompositionMatrix( p, n, r )`

Here, p is the characteristic and $S(n,r)$ is the Schur algebra in question, whose modules are the homogeneous polynomial modules for GL_n in degree `r`. The rows and columns of the matrix are indexed by the partitions produced by `BoundedPartitions(n,r)`.

Example

```
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 ] ]
```

4.5 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 method is quite slow. People needing such numbers for large partitions should use other methods.

`SymmetricGroupDecompositionNumbers( p, mu )`

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

Example

```
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 ]
```

One can also compute the decomposition matrix for a symmetric group in positive characteristic.

`SymmetricGroupDecompositionMatrix( p, n )`

This returns the decomposition matrix for the symmetric group on `n` letters, in characteristic p. The rows of the matrix are labeled by the partitions of `n` (in the order produced by `AllPartitions`) and the columns are labeled by p-restricted partitions of `n`.

Example

```
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 ] ]
```

Note that 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 given in `AllPartitions`.

4.6 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 is the dual simple module labeled by λ.

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

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

```
Mullineux( p, mu )
```

This returns the partition $Mull(\mu)$ corresponding to a given p-regular partition μ under the Mullineux map.

Example

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

`pRegularPartitions(p,n)` returns a list of the p-regular partitions of n, in bijection with the list of p-restricted partitions of n produced by `pRestrictedPartitions(p,n)`, using the bijection $\lambda \to Mull(\lambda')$. 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.

Finally, we have a function that computes the conjugate of a partition.

```
Conjugate( lambda )
```

This returns the conjugate (or transpose) of the input partition.

Example

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

Chapter 5

Case studies

In this chapter we give a few examples of the package in use, by obtaining information about certain Weyl modules. This is intended to illustrate some of the possibilities and demonstrate how the commands can work together.

5.1 Type G 2 in characteristic 2

In this section we study a few small Weyl modules for type G_2 in characteristic 2. We will show that the Weyl module of highest weight $[2,0]$ has four composition factors and is non-rigid (meaning that its socle series does not coincide with its radical series).

5.1.1 Dimensions of restricted simples

There are four restricted weights in this situation: $[0,0]$, $[1,0]$, $[0,1]$, and $[1,1]$. Let's start by computing the dimensions of the Weyl modules with those highest weights.

Example

```
gap> V00 := WeylModule(2,[0,0],"G",2); Dim(V00);
<Type G2 Weyl module of highest weight [ 0, 0 ] at prime p = 2>
1
gap> V10:=WeylModule(2,[1,0],"G",2); Dim(V10);
<Type G2 Weyl module of highest weight [ 1, 0 ] at prime p = 2>
7
gap> V01:=WeylModule(2,[0,1],"G",2); Dim(V01);
<Type G2 Weyl module of highest weight [ 0, 1 ] at prime p = 2>
14
gap> V11:=WeylModule(2,[1,1],"G",2); Dim(V11);
<Type G2 Weyl module of highest weight [ 1, 1 ] at prime p = 2>
64
```

Of course we know that `V00` is the trivial module and `V11` is the Steinberg module; they are both simple. It turns out that `V01` is also simple but `V10` is not.

Example

```
gap> MaximalVectors(V10); List(last,Weight);
[ 1*v0, y4*v0 ]
[ [ 1, 0 ], [ 0, 0 ] ]
gap> MaximalVectors(V01); List(last,Weight);
```

```
[ 1*v0 ]
[ [ 0, 1 ] ]
```

This suggests that V10 is a non-split extension of its simple head by the trivial module. If so, the simple module of highest weight $[1,0]$ has dimension 6. We can double check that guess, as follows.

Example

```
gap> S := MaximalSubmodule(V10); Dim(S);
<SubWeylModule of dimension 1, generated by elements [ y4*v0 ] of weights
[ [ 0, 0 ] ]>, in
<Type G2 Weyl module of highest weight [ 1, 0 ] at prime p = 2>
1
```

This confirms our guess. The simple head of V10 must have dimension 6. Another way to confirm it is to compute the simple character of highest weight $[1,0]$.

Example

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

Yet another way to confirm this is the following.

Example

```
gap> Dim(SimpleQuotient(V10));
6
```

At this point, we know the characters of all the restricted simples, and thus by Steinberg's tensor product theorem, we know the characters of all the simple moduless for type G_2 in characteristc 2. For example,

Example

```
gap> SimpleCharacter(2,[3,0],"G",2);
[ [ 3, 0 ], 1, [ -1, 2 ], 1, [ 5, -2 ], 1, [ -3, 2 ], 2, [ 3, -2 ], 2,
  [ -1, 0 ], 1, [ 1, 1 ], 1, [ -3, 3 ], 1, [ 3, -1 ], 2, [ -5, 3 ], 1,
  [ 1, -1 ], 1, [ -3, 1 ], 2, [ 4, -1 ], 1, [ 0, 1 ], 2, [ 6, -3 ], 1,
  [ -2, 1 ], 1, [ 4, -3 ], 1, [ 0, -1 ], 2, [ -4, 3 ], 1, [ 2, -1 ], 1,
  [ -6, 3 ], 1, [ -4, 1 ], 1, [ -1, 1 ], 1, [ 5, -3 ], 1, [ 3, -3 ], 1,
  [ -1, -1 ], 1, [ 1, 0 ], 1, [ -5, 2 ], 1, [ 1, -2 ], 1, [ -3, 0 ], 1 ]
```

5.1.2 Structure of the Weyl module of highest weight $[2,0]$

Now we turn to an analysis of the structure of the Weyl module of highest weight $[2,0]$.

Example

```
gap> V20 := WeylModule(2,[2,0],"G",2); Dim(V20);
<Type G2 Weyl module of highest weight [ 2, 0 ] at prime p = 2>
27
```

The module has dimension 27. Let's see how many maximal vectors it has.

Example
```
gap> m20 := MaximalVectors(V20); List(m20,Weight);
[ 1*v0, y1*v0, y4*v0 ]
[ [ 2, 0 ], [ 0, 1 ], [ 1, 0 ] ]
```

and compute the submodules generated by the maximal vectors different from the generator of V20.

Example
```
gap> S1 := SubWeylModule(V20,m20[2]); S2 := SubWeylModule(V20,m20[3]);
<SubWeylModule of dimension 14, generated by elements [ y1*v0 ] of weights
[ [ 0, 1 ] ]>, in
<Type G2 Weyl module of highest weight [ 2, 0 ] at prime p = 2>
<SubWeylModule of dimension 6, generated by elements [ y4*v0 ] of weights
[ [ 1, 0 ] ]>, in
<Type G2 Weyl module of highest weight [ 2, 0 ] at prime p = 2>
```

Both submodules are simple, hence lie in the socle. In fact, we have

Example
```
gap> soc20 := SocleWeyl(V20);
<SubWeylModule of dimension 20, generated by elements
[ y1*v0, y4*v0 ] of weights [ [ 0, 1 ], [ 1, 0 ] ]>, in
<Type G2 Weyl module of highest weight [ 2, 0 ] at prime p = 2>
```

which confirms this.

The corresponding quotient by the socle is thus a 7-dimensional module. It would be useful to compute the socle of this quotient.

Example
```
gap> SocleWeyl(QuotientWeylModule(soc20));
<SubQuotient WeylModule of dimension 1, generated by elements
[ y4^(2)*v0 ] of weights [ [ 0, 0 ] ]>, in
<Quotient of Type G2 Weyl module of highest weight [ 2, 0 ] at prime p = 2>
```

We see that the socle is of dimension 1, isomorphic to the trivial module. Since the dimension of the simple module of highest weight $[2,0]$ is equal to 6, as it is the Frobenius twist of the simple module of highest weight $[1,0]$, we conclude that this quotient module is uniserial of length two. In fact, it is isomorphic to the Frobenius twist of V10.

Computing the socle series bears this out.

Example
```
gap> SocleSeries(V20);
[ <SubWeylModule of dimension 20, generated by elements
    [ y1*v0, y4*v0 ] of weights [ [ 0, 1 ], [ 1, 0 ] ]>, in
    <Type G2 Weyl module of highest weight [ 2, 0 ] at prime p = 2>,
  <SubWeylModule of dimension 21, generated by elements
    [ y1*v0, y4*v0, y4^(2)*v0 ] of weights [ [ 0, 1 ], [ 1, 0 ], [ 0, 0 ]
     ]>, in
    <Type G2 Weyl module of highest weight [ 2, 0 ] at prime p = 2>,
  <SubWeylModule of dimension 27, generated by elements
    [ y1*v0, y4*v0, y4^(2)*v0, 1*v0 ] of weights
    [ [ 0, 1 ], [ 1, 0 ], [ 0, 0 ], [ 2, 0 ] ]>, in
    <Type G2 Weyl module of highest weight [ 2, 0 ] at prime p = 2> ]
```

and computing the decomposition numbers confirms that this Weyl module has four composition factors.

Example

```
gap> DecompositionNumbers(V20);
[ [ 2, 0 ], 1, [ 0, 1 ], 1, [ 1, 0 ], 1, [ 0, 0 ], 1 ]
```

The complete submodule structure is still unclear, however. We need more data. Let's look at a couple of other quotients of V20. Recall that we previously computed S1 and S2 as the simple modules, of dimension 14 and 6 respectively, whose direct sum makes up the socle of V20.

Example

```
gap> Q1 := QuotientWeylModule(S1); Q2 := QuotientWeylModule(S2);
<Quotient of Type G2 Weyl module of highest weight [ 2, 0 ] at prime p = 2>
<Quotient of Type G2 Weyl module of highest weight [ 2, 0 ] at prime p = 2>
gap> max1 := MaximalVectors(Q1); List(max1,Weight);
[ 1*v0, y4*v0 ]
[ [ 2, 0 ], [ 1, 0 ] ]
gap> max2 := MaximalVectors(Q2); List(max2,Weight);
[ 1*v0, y1*v0, y1*y6*v0+y3*y5*v0+y4^(2)*v0 ]
[ [ 2, 0 ], [ 0, 1 ], [ 0, 0 ] ]
```

Each quotient module has exactly three composition factors. The first, Q1, must be uniserial with a copy of the trivial module in the middle socle layer (otherwise there would be another maximal vector). On the other hand, Q2 has two factors in its socle, of highest weight $[1,0]$ and $[0,0]$. We can check these conclusions by computing the socle series for each quotient.

Example

```
gap> SocleSeries(Q1);
[ <SubQuotient WeylModule of dimension 6, generated by elements
    [ y4*v0 ] of weights [ [ 1, 0 ] ]>, in
    <Quotient of Type G2 Weyl module of highest weight [ 2, 0 ] at prime p =
    2>, <SubQuotient WeylModule of dimension 7, generated by elements
    [ y4*v0, y4^(2)*v0 ] of weights [ [ 1, 0 ], [ 0, 0 ] ]>, in
    <Quotient of Type G2 Weyl module of highest weight [ 2, 0 ] at prime p =
    2>, <SubQuotient WeylModule of dimension 13, generated by elements
    [ y4*v0, y4^(2)*v0, 1*v0 ] of weights [ [ 1, 0 ], [ 0, 0 ], [ 2, 0 ]
     ]>, in
    <Quotient of Type G2 Weyl module of highest weight [ 2, 0 ] at prime p =
    2> ]
```

Example

```
gap> SocleSeries(Q2);
[ <SubQuotient WeylModule of dimension 15, generated by elements
    [ y1*v0, y1*y6*v0+y3*y5*v0+y4^(2)*v0 ] of weights [ [ 0, 1 ], [ 0, 0 ]
     ]>, in
    <Quotient of Type G2 Weyl module of highest weight [ 2, 0 ] at prime p =
    2>, <SubQuotient WeylModule of dimension 21, generated by elements
    [ y1*v0, y1*y6*v0+y3*y5*v0+y4^(2)*v0, 1*v0 ] of weights
    [ [ 0, 1 ], [ 0, 0 ], [ 2, 0 ] ]>, in
    <Quotient of Type G2 Weyl module of highest weight [ 2, 0 ] at prime p =
    2> ]
```

Next we construct the submodule of V20 generated by the inverse image of the weight zero socle generator in the quotient Q2.

Example

```
gap> S3 := SubWeylModule(V20,max2[3]);
<SubWeylModule of dimension 7, generated by elements
[ y1*y6*v0+y3*y5*v0+y4^(2)*v0 ] of weights [ [ 0, 0 ] ]>, in
<Type G2 Weyl module of highest weight [ 2, 0 ] at prime p = 2>
gap> maxS3 := MaximalVectors(S3); List(maxS3,Weight);
[ y4*v0 ]
[ [ 1, 0 ] ]
```

This submodule has only one maximal vector, which means that it has a simple socle of highest weight $[1,0]$. Hence S3 is an extension of that simple by the trivial module. It is interesting to construct the corresponding quotient of the ambient Weyl module.

Example

```
gap> M := QuotientWeylModule(S3); Dim(M);
<Quotient of Type G2 Weyl module of highest weight [ 2, 0 ] at prime p = 2>
20
```

This quotient has dimension 20, and should have a simple socle of highest weight $[0,1]$, of dimesnion 14.

Example

```
gap> maxM := MaximalVectors(M); List(maxM,Weight);
[ 1*v0, y1*v0 ]
[ [ 2, 0 ], [ 0, 1 ] ]
gap> SocleSeries(M);
[ <SubQuotient WeylModule of dimension 14, generated by elements
    [ y1*v0 ] of weights [ [ 0, 1 ] ]>, in
    <Quotient of Type G2 Weyl module of highest weight [ 2, 0 ] at prime p =
    2>, <SubQuotient WeylModule of dimension 20, generated by elements
    [ y1*v0, 1*v0 ] of weights [ [ 0, 1 ], [ 2, 0 ] ]>, in
    <Quotient of Type G2 Weyl module of highest weight [ 2, 0 ] at prime p =
    2> ]
```

This confirms that M is uniserial length two, in other words an extension of the simple of highest weight $[0,1]$ by the simple of highest weight $[2,0]$. The latter simple is isomorphic to the Frobenius twist of the simple of highest weight $[1,0]$.

We conclude from the above calcuations that V20 has second radical layer with composition factors of highest weight $[0,0]$ and $[0,1]$. However, the second socle layer is simple of highest weight $[0,0]$. Thus, we see that V20 is not rigid (its socle series differs from its radical series).

The module M appears in Section 3.7 of [BNPS20], constructed differently. It plays an important role in the counterexample to Donkin's tilting module conjecture produced in that paper.

5.1.3 The socle of the Weyl module of highest weight $[2,2]$

The key conclusion of the calculations in [BNPS20] is that the Weyl module of highest weight $[2,2]$ has a non-simple socle, with two composition factors of highest weights $[0,0]$ and $[0,1]$. We can verify that conclusion using GAP.

Example
```
gap> V22 := WeylModule(2,[2,2],"G",2);
<Type G2 Weyl module of highest weight [ 2, 2 ] at prime p = 2>
gap> Dim(V22);
729
gap> SocleWeyl(V22);
***** WARNING: Ambiguous module detected *****
<SubWeylModule of dimension 15, generated by elements
[ y1*y2*y3*y4*y5*v0+y1*y2*y4^(3)*v0+y1*y3*y4*y6*v0, y1*y2*y3*y4*y5*y6*v0
 ] of weights [ [ 0, 1 ], [ 0, 0 ] ]>, in
<Type G2 Weyl module of highest weight [ 2, 2 ] at prime p = 2>
```

This is the same as the socle of the module `Q2` in the preceding subsection. Notice that `V22` is ambiguous (it has at least two independent maximal vectors of the same weight). In fact, we can find what they look like.

Example
```
gap> max := MaximalVectors(V22); List(max,Weight);
[ 1*v0, y1*v0, y2*v0, y1^(2)*y2*v0, y1*y2*y3*v0, y2*y4*v0, y1*y2*y4*v0,
  y1*y2*y5*v0+y1*y3*y4*v0+y1*y6*v0, y1*y2*y3*y4*v0+y1*y2*y6*v0+y2*y3*y5*v0+y
    3*y6*v0, y1*y2*y4*y6*v0+y2*y3*y4*y5*v0+y2*y4^(3)*v0+y3*y4*y6*v0,
  y1*y2*y3*y4*y5*v0+y1*y2*y4^(3)*v0+y1*y3*y4*y6*v0,
  y1*y2*y3^(2)*y4*y5*v0+y2*y4*y5*y6*v0+y3^(3)*y4*y5*v0, y1*y2*y3*y4*y5*y6*v0 ]
[ [ 2, 2 ], [ 0, 3 ], [ 5, 0 ], [ 1, 2 ], [ 4, 0 ], [ 4, 0 ], [ 2, 1 ],
  [ 0, 2 ], [ 3, 0 ], [ 2, 0 ], [ 0, 1 ], [ 1, 0 ], [ 0, 0 ] ]
```

So the ambiguous maximal vectors are the ones of weight $[4,0]$. Notice that there is also a maximal vector of weight $[2,0]$. It must generate a homomorphic image of the Weyl module `V20` from the preceding subsection.

Example
```
gap> vec := max[10]; Weight(vec);
y1*y2*y4*y6*v0+y2*y3*y4*y5*v0+y2*y4^(3)*v0+y3*y4*y6*v0
[ 2, 0 ]
gap> S := SubWeylModule(V22,vec);
<SubWeylModule of dimension 20, generated by elements
[ y1*y2*y4*y6*v0+y2*y3*y4*y5*v0+y2*y4^(3)*v0+y3*y4*y6*v0 ] of weights
[ [ 2, 0 ] ]>, in
<Type G2 Weyl module of highest weight [ 2, 2 ] at prime p = 2>
gap> char := Character(S);
[ [ 2, 0 ], 1, [ 0, 1 ], 1, [ 3, -1 ], 1, [ -1, 1 ], 1, [ 2, -1 ], 1,
  [ -2, 2 ], 1, [ 4, -2 ], 1, [ 0, 0 ], 2, [ -4, 2 ], 1, [ 2, -2 ], 1,
  [ -2, 0 ], 1, [ 1, 0 ], 1, [ -3, 2 ], 1, [ -2, 1 ], 1, [ 0, -1 ], 1,
  [ 3, -2 ], 1, [ 1, -1 ], 1, [ -3, 1 ], 1, [ -1, 0 ], 1 ]
gap> DecomposeCharacter(char,2,"G",2);
[ [ 2, 0 ], 1, [ 0, 1 ], 1 ]
```

The above shows that the homomorphic image is isomorphic to the module `M` construced in the preceding subsection.

5.2 Type A 2

We analyze the structure of some Weyl modules in type A_2.

5.2.1 The Weyl module of highest weight $[7,7]$ in characteristic 2

Example

```
gap> V77 := WeylModule(2,[7,7],"A",2);
<Type A2 Weyl module of highest weight [ 7, 7 ] at prime p = 2>
gap> DecompositionNumbers(V77);
[ [ 7, 7 ], 1 ]
```

In characteristic 2 this is a simple Weyl module. In fact it is a Steinberg module.

5.2.2 The Weyl module of highest weight $[7,7]$ in characteristic 3

Example

```
gap> V77 := WeylModule(3,[7,7],"A",2);
<Type A2 Weyl module of highest weight [ 7, 7 ] at prime p = 3>
gap> DecompositionNumbers(V77);
[ [ 7, 7 ], 1, [ 6, 6 ], 1, [ 3, 9 ], 1, [ 9, 3 ], 1, [ 4, 7 ], 1, [ 7, 4 ],
  1, [ 3, 6 ], 2, [ 6, 3 ], 2, [ 3, 3 ], 1, [ 0, 0 ], 1 ]
gap> Length(last);
20
```

In characteristic 3 this Weyl module has 10 composition factors, two of which have multiplicity 2. This decomposition pattern is not generic. The multiplicities are explained by general results in the paper [DS87].

5.2.3 The Weyl module of highest weight $[7,7]$ in characteristic 5

This Weyl module illustrates one of Jantzen's generic decomposition patterns when the characteristic p is sufficiently large and the highest weight is in the lowest p^2 alcove and sufficiently far away from its walls.

Example

```
gap> V77 := WeylModule(5,[7,7],"A",2);
<Type A2 Weyl module of highest weight [ 7, 7 ] at prime p = 5>
gap> Dim(V77);
512
gap> DecompositionNumbers(V77); Length(last);
[ [ 7, 7 ], 1, [ 6, 6 ], 1, [ 1, 10 ], 1, [ 10, 1 ], 1, [ 2, 8 ], 1,
  [ 8, 2 ], 1, [ 0, 6 ], 1, [ 6, 0 ], 1, [ 1, 1 ], 1 ]
18
```

This Weyl module has nine composition factors, each of multiplicity one. Let's compute its maximal vectors.

Example

```
gap> max77 := MaximalVectors(V77); List(last,Weight);
[ 1*v0, -2*y1*y2*v0+y3*v0, y1^(3)*v0, y2^(3)*v0, -1*y1^(2)*y3*v0+y1^(3)*y2*v0,
  y1*y2^(3)*v0, y1^(3)*y2^(3)*y3^(3)*v0+y1^(4)*y2^(4)*y3^(2)*v0 ]
[ [ 7, 7 ], [ 6, 6 ], [ 1, 10 ], [ 10, 1 ], [ 2, 8 ], [ 8, 2 ], [ 1, 1 ] ]
```

There are seven maximal vectors. Now we compute the socle series. We are going to do it one step at a time, instead of using the `SocleSeries` command.

Example

```
gap> soc1 := SocleWeyl(V77);
<SubWeylModule of dimension 8, generated by elements
[ y1^(3)*y2^(3)*y3^(3)*v0+y1^(4)*y2^(4)*y3^(2)*v0 ] of weights
[ [ 1, 1 ] ]>, in
<Type A2 Weyl module of highest weight [ 7, 7 ] at prime p = 5>
```

The socle is simple, of highest weight $[1,1]$. Now let's compute the next socle.

Example

```
gap> soc2 := NextSocle(soc1);
<SubWeylModule of dimension 242, generated by elements
[ y1^(3)*y2^(3)*y3^(3)*v0+y1^(4)*y2^(4)*y3^(2)*v0,
  -1*y1^(2)*y3*v0+y1^(3)*y2*v0, y1*y2^(3)*v0 ] of weights
[ [ 1, 1 ], [ 2, 8 ], [ 8, 2 ] ]>, in
<Type A2 Weyl module of highest weight [ 7, 7 ] at prime p = 5>
```

This adds two more simples, of highest weight $[2,8]$ and $[8,2]$.

Example

```
gap> soc3 := NextSocle(soc2);
<SubWeylModule of dimension 360, generated by elements
[ y1^(3)*y2^(3)*y3^(3)*v0+y1^(4)*y2^(4)*y3^(2)*v0,
  -1*y1^(2)*y3*v0+y1^(3)*y2*v0, y1*y2^(3)*v0, -2*y1*y2*v0+y3*v0, y1^(3)*v0,
  y2^(3)*v0, -2*y1^(2)*y3^(3)*v0+y1^(5)*y2^(3)*v0,
  y1^(3)*y2^(5)*v0+y2^(2)*y3^(3)*v0 ] of weights
[ [ 1, 1 ], [ 2, 8 ], [ 8, 2 ], [ 6, 6 ], [ 1, 10 ], [ 10, 1 ], [ 0, 6 ],
  [ 6, 0 ] ]>, in
<Type A2 Weyl module of highest weight [ 7, 7 ] at prime p = 5>
```

This socle adds 5 additional simples, as indicated.

Example

```
gap> soc4 := NextSocle(soc3);
<SubWeylModule of dimension 512, generated by elements
[ y1^(3)*y2^(3)*y3^(3)*v0+y1^(4)*y2^(4)*y3^(2)*v0,
  -1*y1^(2)*y3*v0+y1^(3)*y2*v0, y1*y2^(3)*v0, -2*y1*y2*v0+y3*v0, y1^(3)*v0,
  y2^(3)*v0, -2*y1^(2)*y3^(3)*v0+y1^(5)*y2^(3)*v0,
  y1^(3)*y2^(5)*v0+y2^(2)*y3^(3)*v0, 1*v0 ] of weights
[ [ 1, 1 ], [ 2, 8 ], [ 8, 2 ], [ 6, 6 ], [ 1, 10 ], [ 10, 1 ], [ 0, 6 ],
  [ 6, 0 ], [ 7, 7 ] ]>, in
<Type A2 Weyl module of highest weight [ 7, 7 ] at prime p = 5>
```

and finally we arrive at the entire module, of dimension 512. So we have computed the socle series.

These results agree with the papers [Irv86] and [DS85] (see also [DS87]). One could continue to verify various aspects of the submodule structure as calculated in those papers.

Chapter 6

Reference: Declarations by File

This chapter documents the declarations by file in the `lib` folder, following the load order specified in the file `init.g`. This is provided mainly to aid in subsequent development efforts.

6.1 Contents of weylmodules.gd

6.1.1 IsWeylModule (for CategoryCollections(IsLeftAlgebraModuleElement))

▷ `IsWeylModule(`*arg*`)` (filter)

Returns: `true` or `false`

6.1.2 WeylModule (for IsPosInt, IsList, IsString, IsPosInt)

▷ `WeylModule(`*arg1*`, `*arg2*`, `*arg3*`, `*arg4*`)` (operation)

6.1.3 WeylModule (for IsWeylModule,IsList)

▷ `WeylModule(`*arg1*`, `*arg2*`)` (operation)

6.1.4 IsAmbiguous (for IsWeylModule)

▷ `IsAmbiguous(`*arg*`)` (operation)

6.1.5 AmbiguousMaxVecs (for IsWeylModule)

▷ `AmbiguousMaxVecs(`*arg*`)` (operation)

6.1.6 TheLieAlgebra (for IsWeylModule)

▷ `TheLieAlgebra(`*arg*`)` (operation)

6.1.7 TheCharacteristic (for IsWeylModule)

▷ TheCharacteristic(*arg*) (operation)

6.1.8 BasisVecs (for IsWeylModule)

▷ BasisVecs(*arg*) (operation)

6.1.9 Generator (for IsWeylModule)

▷ Generator(*arg*) (operation)

6.1.10 Dim (for IsWeylModule)

▷ Dim(*arg*) (operation)

6.1.11 Weights (for IsWeylModule)

▷ Weights(*arg*) (operation)

6.1.12 DominantWeights (for IsWeylModule)

▷ DominantWeights(*arg*) (operation)

6.1.13 WeightSpace (for IsWeylModule,IsList)

▷ WeightSpace(*arg1*, *arg2*) (operation)

6.1.14 WeightSpaces (for IsWeylModule)

▷ WeightSpaces(*arg*) (operation)

6.1.15 ActOn (for IsWeylModule, IsUEALatticeElement, IsLeftAlgebraModuleElement)

▷ ActOn(*arg1*, *arg2*, *arg3*) (operation)

6.1.16 DominantWeightSpaces (for IsWeylModule)

▷ DominantWeightSpaces(*arg*) (operation)

6.1.17 Character (for IsWeylModule)

▷ `Character(arg)` (operation)

6.1.18 SimpleLieAlgGens (for IsWeylModule)

▷ `SimpleLieAlgGens(arg)` (operation)

6.1.19 IsMaximalVector (for IsWeylModule,IsLeftAlgebraModuleElement)

▷ `IsMaximalVector(arg1, arg2)` (operation)

6.1.20 MaximalVectors (for IsWeylModule,IsList)

▷ `MaximalVectors(arg1, arg2)` (operation)

6.1.21 MaximalVectors (for IsWeylModule)

▷ `MaximalVectors(arg)` (operation)

6.1.22 SocleSeries (for IsWeylModule)

▷ `SocleSeries(arg)` (operation)

6.1.23 SimpleQuotient (for IsWeylModule)

▷ `SimpleQuotient(arg)` (operation)

6.1.24 MaximalSubmodule (for IsWeylModule)

▷ `MaximalSubmodule(arg)` (operation)

6.1.25 DecompositionNumbers (for IsWeylModule)

▷ `DecompositionNumbers(arg)` (operation)

6.2 Contents of submodules.gd

6.2.1 IsSubWeylModule (for CategoryCollections(IsLeftAlgebraModuleElement))

▷ `IsSubWeylModule(arg)` (filter)

Returns: `true` or `false`

6.2.2 SubWeylModule (for IsWeylModule,IsLeftAlgebraModuleElement)

▷ `SubWeylModule(arg1, arg2)` (operation)

6.2.3 SubWeylModule (for IsSubWeylModule,IsLeftAlgebraModuleElement)

▷ `SubWeylModule(arg1, arg2)` (operation)

6.2.4 SubWeylModule (for IsWeylModule,IsList)

▷ `SubWeylModule(arg1, arg2)` (operation)

6.2.5 SubWeylModuleDirectSum (for IsWeylModule,IsList)

▷ `SubWeylModuleDirectSum(arg1, arg2)` (operation)

6.2.6 IsAmbiguous (for IsSubWeylModule)

▷ `IsAmbiguous(arg)` (operation)

6.2.7 AmbiguousMaxVecs (for IsSubWeylModule)

▷ `AmbiguousMaxVecs(arg)` (operation)

6.2.8 Generators (for IsSubWeylModule)

▷ `Generators(arg)` (operation)

6.2.9 BasisVecs (for IsSubWeylModule)

▷ `BasisVecs(arg)` (operation)

6.2.10 Dim (for IsSubWeylModule)

▷ Dim(*arg*) (operation)

6.2.11 AmbientWeylModule (for IsSubWeylModule)

▷ AmbientWeylModule(*arg*) (operation)

6.2.12 Weights (for IsSubWeylModule)

▷ Weights(*arg*) (operation)

6.2.13 DominantWeights (for IsSubWeylModule)

▷ DominantWeights(*arg*) (operation)

6.2.14 WeightSpaces (for IsSubWeylModule)

▷ WeightSpaces(*arg*) (operation)

6.2.15 Character (for IsSubWeylModule)

▷ Character(*arg*) (operation)

6.2.16 DominantWeightSpaces (for IsSubWeylModule)

▷ DominantWeightSpaces(*arg*) (operation)

6.2.17 WeightSpace (for IsSubWeylModule,IsList)

▷ WeightSpace(*arg1, arg2*) (operation)

6.2.18 IsMaximalVector (for IsSubWeylModule,IsLeftAlgebraModuleElement)

▷ IsMaximalVector(*arg1, arg2*) (operation)

6.2.19 MaximalVectors (for IsSubWeylModule,IsList)

▷ MaximalVectors(*arg1, arg2*) (operation)

6.2.20 MaximalVectors (for IsSubWeylModule)

▷ MaximalVectors(*arg*) (operation)

6.2.21 IsWithin (for IsSubWeylModule,IsLeftAlgebraModuleElement)

▷ IsWithin(*arg1, arg2*) (operation)

6.2.22 SocleWeyl (for IsWeylModule)

▷ SocleWeyl(*arg*) (operation)

6.2.23 NextSocle (for IsSubWeylModule)

▷ NextSocle(*arg*) (operation)

6.2.24 GensNextSocle (for IsSubWeylModule)

▷ GensNextSocle(*arg*) (operation)

6.3 Contents of characters.gd

6.3.1 Weight (for IsLeftAlgebraModuleElement)

▷ Weight(*arg*) (operation)

6.3.2 DifferenceCharacter (for IsList,IsList)

▷ DifferenceCharacter(*arg1, arg2*) (operation)

6.3.3 ProductCharacter (for IsList,IsList)

▷ ProductCharacter(*arg1, arg2*) (operation)

6.3.4 DecomposeCharacter (for IsList, IsPosInt,IsString, IsPosInt)

▷ DecomposeCharacter(*arg1, arg2, arg3, arg4*) (operation)

6.3.5 SimpleCharacter (for IsPosInt, IsList, IsString, IsPosInt)

▷ SimpleCharacter(*arg1, arg2, arg3, arg4*) (operation)

6.3.6 SimpleCharacter (for IsWeylModule, IsList)

▷ SimpleCharacter(*arg1, arg2*) (operation)

6.4 Contents of partitions.gd

6.4.1 CompositionToWeight (for IsList)

▷ CompositionToWeight(*arg*) (operation)

6.4.2 WeightToComposition (for IsInt, IsList)

▷ WeightToComposition(*arg1, arg2*) (operation)

6.4.3 BoundedPartitions (for IsInt, IsInt, IsInt)

▷ BoundedPartitions(*arg1, arg2, arg3*) (operation)

6.4.4 BoundedPartitions (for IsInt, IsInt)

▷ BoundedPartitions(*arg1, arg2*) (operation)

6.4.5 pRestrictedPartitions (for IsInt, IsInt)

▷ pRestrictedPartitions(*arg1, arg2*) (operation)

6.4.6 AllPartitions (for IsInt)

▷ AllPartitions(*arg*) (operation)

6.4.7 Conjugate (for IsList)

▷ Conjugate(*arg*) (operation)

6.4.8 pRestricted (for IsPosInt, IsList)

▷ pRestricted(*arg1, arg2*) (operation)

6.4.9 pRegular (for IsPosInt, IsList)

▷ pRegular(*arg1, arg2*) (operation)

6.4.10 Mullineux (for IsPosInt, IsList)

▷ Mullineux(*arg1, arg2*) (operation)

6.4.11 pRegularPartitions (for IsPosInt, IsPosInt)

▷ pRegularPartitions(*arg1, arg2*) (operation)

6.5 Contents of quotient.gd

6.5.1 IsQuotientWeylModule (for CategoryCollections(IsLeftAlgebraModuleElement))

▷ IsQuotientWeylModule(*arg*) (filter)
Returns: true or false

6.5.2 QuotientWeylModule (for IsSubWeylModule)

▷ QuotientWeylModule(*arg*) (operation)

6.5.3 AmbientWeylModule (for IsQuotientWeylModule)

▷ AmbientWeylModule(*arg*) (operation)

6.5.4 DefiningKernel (for IsQuotientWeylModule)

▷ DefiningKernel(*arg*) (operation)

6.5.5 IsAmbiguous (for IsQuotientWeylModule)

▷ IsAmbiguous(*arg*) (operation)

6.5.6 AmbiguousMaxVecs (for IsQuotientWeylModule)

▷ AmbiguousMaxVecs(*arg*) (operation)

6.5.7 TheLieAlgebra (for IsQuotientWeylModule)

▷ TheLieAlgebra(*arg*) (operation)

6.5.8 TheCharacteristic (for IsQuotientWeylModule)

▷ TheCharacteristic(*arg*) (operation)

6.5.9 BasisVecs (for IsQuotientWeylModule)

▷ BasisVecs(*arg*) (operation)

6.5.10 Generator (for IsQuotientWeylModule)

▷ Generator(*arg*) (operation)

6.5.11 Dim (for IsQuotientWeylModule)

▷ Dim(*arg*) (operation)

6.5.12 Weights (for IsQuotientWeylModule)

▷ Weights(*arg*) (operation)

6.5.13 DominantWeights (for IsQuotientWeylModule)

▷ DominantWeights(*arg*) (operation)

6.5.14 WeightSpaces (for IsQuotientWeylModule)

▷ WeightSpaces(*arg*) (operation)

6.5.15 Character (for IsQuotientWeylModule)

▷ Character(*arg*) (operation)

6.5.16 DominantWeightSpaces (for IsQuotientWeylModule)

▷ DominantWeightSpaces(*arg*) (operation)

6.5.17 WeightSpace (for IsQuotientWeylModule,IsList)

▷ WeightSpace(*arg1*, *arg2*) (operation)

6.5.18 ActOn (for IsQuotientWeylModule,IsUEALatticeElement,IsLeftAlgebraModuleElement)

▷ ActOn(*arg1*, *arg2*, *arg3*) (operation)

6.5.19 MaximalVectors (for IsQuotientWeylModule,IsList)

▷ MaximalVectors(*arg1*, *arg2*) (operation)

6.5.20 MaximalVectors (for IsQuotientWeylModule)

▷ MaximalVectors(*arg*) (operation)

6.5.21 SocleSeries (for IsQuotientWeylModule)

▷ SocleSeries(*arg*) (operation)

6.6 Contents of schuralgebras.gd

6.6.1 IsSchurAlgebraWeylModule (for IsWeylModule)

▷ IsSchurAlgebraWeylModule(*arg*) (filter)
Returns: true or false

6.6.2 SchurAlgebraWeylModule (for IsInt, IsList)

▷ SchurAlgebraWeylModule(*arg1*, *arg2*) (operation)

6.6.3 SchurAlgebraDecompositionMatrix (for IsInt, IsInt, IsInt)

▷ SchurAlgebraDecompositionMatrix(*arg1*, *arg2*, *arg3*) (operation)

6.6.4 SymmetricGroupDecompositionNumbers (for IsInt, IsList)

▷ SymmetricGroupDecompositionNumbers(*arg1, arg2*) (operation)

6.6.5 SymmetricGroupDecompositionMatrix (for IsInt, IsInt)

▷ SymmetricGroupDecompositionMatrix(*arg1, arg2*) (operation)

6.7 Contents of subquotient.gd

6.7.1 IsSubQuotientWeylModule (for CategoryCollections(IsLeftAlgebraModuleElement))

▷ IsSubQuotientWeylModule(*arg*) (filter)

Returns: true or false

6.7.2 IsWithin (for IsSubQuotientWeylModule,IsLeftAlgebraModuleElement)

▷ IsWithin(*arg1, arg2*) (operation)

6.7.3 SubWeylModule (for IsQuotientWeylModule,IsLeftAlgebraModuleElement)

▷ SubWeylModule(*arg1, arg2*) (operation)

6.7.4 SubWeylModule (for IsSubQuotientWeylModule,IsLeftAlgebraModuleElement)

▷ SubWeylModule(*arg1, arg2*) (operation)

6.7.5 SubWeylModule (for IsQuotientWeylModule,IsList)

▷ SubWeylModule(*arg1, arg2*) (operation)

6.7.6 SubWeylModuleDirectSum (for IsQuotientWeylModule,IsList)

▷ SubWeylModuleDirectSum(*arg1, arg2*) (operation)

6.7.7 Generators (for IsSubQuotientWeylModule)

▷ Generators(*arg*) (operation)

6.7.8 BasisVecs (for IsSubQuotientWeylModule)

▷ `BasisVecs(arg)` (operation)

6.7.9 Dim (for IsSubQuotientWeylModule)

▷ `Dim(arg)` (operation)

6.7.10 AmbientQuotient (for IsSubQuotientWeylModule)

▷ `AmbientQuotient(arg)` (operation)

6.7.11 Weights (for IsSubQuotientWeylModule)

▷ `Weights(arg)` (operation)

6.7.12 WeightSpaces (for IsSubQuotientWeylModule)

▷ `WeightSpaces(arg)` (operation)

6.7.13 Character (for IsSubQuotientWeylModule)

▷ `Character(arg)` (operation)

6.7.14 DominantWeightSpaces (for IsSubQuotientWeylModule)

▷ `DominantWeightSpaces(arg)` (operation)

6.7.15 WeightSpace (for IsSubQuotientWeylModule,IsList)

▷ `WeightSpace(arg1, arg2)` (operation)

6.7.16 SocleWeyl (for IsQuotientWeylModule)

▷ `SocleWeyl(arg)` (operation)

6.7.17 NextSocle (for IsSubQuotientWeylModule)

▷ `NextSocle(arg)` (operation)

References

[BDM11] C. Bowman, S. R. Doty, and S. Martin. Decomposition of tensor products of modular irreducible representations for SL_3. *Int. Electron. J. Algebra*, 9:177–219, 2011. With an appendix by C. M. Ringel. 5

[BNPS20] Christopher P. Bendel, Daniel K. Nakano, Cornelius Pillen, and Paul Sobaje. Counterexamples to the tilting and (p,r)-filtration conjectures. *J. Reine Angew. Math.*, 767:193–202, 2020. 5, 29

[CPS75] Edward Cline, Brian Parshall, and Leonard Scott. Cohomology of finite groups of Lie type. I. *Inst. Hautes Études Sci. Publ. Math.*, (45):169–191, 1975. 6, 11

[DS85] Stephen R. Doty and John B. Sullivan. The submodule structure of Weyl modules for SL_3. *J. Algebra*, 96(1):78–93, 1985. 32

[DS87] Stephen R. Doty and John B. Sullivan. Filtration patterns for representations of algebraic groups and their Frobenius kernels. *Math. Z.*, 195(3):391–407, 1987. 31, 32

[Gre07] J. A. Green. *Polynomial representations of* GL_n, volume 830 of *Lecture Notes in Mathematics*. Springer, Berlin, augmented edition, 2007. With an appendix on Schensted correspondence and Littelmann paths by K. Erdmann, Green and M. Schocker. 23

[Irv86] Ronald S. Irving. The structure of certain highest weight modules for SL_3. *J. Algebra*, 99(2):438–457, 1986. 5, 32

[Jam78] G. D. James. *The representation theory of the symmetric groups*, volume 682 of *Lecture Notes in Mathematics*. Springer, Berlin, 1978. 24

[Jan03] Jens Carsten Jantzen. *Representations of algebraic groups*, volume 107 of *Mathematical Surveys and Monographs*. American Mathematical Society, Providence, RI, second edition, 2003. 5

[Xi99] Nanhua Xi. Maximal and primitive elements in Weyl modules for type A_2. *J. Algebra*, 215(2):735–756, 1999. 5

Index