WeylModules

for simple simply-connected algebraic groups

2.1

21 June 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.

Contents

Chapter 1

Introduction

1.1 Purpose

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.

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.

1.2 Counterexample to Donkin's conjecture

In 2019, Chris Bendel, Dan Nakano, Cornelius Pillen, and Paul Sobaje [BNPS20] found the first counterexample to Donkin's tilting module conjecture using this package. This important advance led to further development of the package.

1.3 ChangeLog

The initial release was Version 1.0 in 2009.

1. Versions 1.0 and 1.1 were released in 2009. The initial development was spurred by work on the paper [BDM11].

2. Version 2.0 was released on 29 February 2024. The `SubmoduleStructure` command was eliminated. Support was added for subquotients.

3. Version 2.1 (this version) was released on 21 June 2024. The documentation was completely rewritten. There is more support for submodules and subquotients, including the new functions: `SocleLengthTwoQuotient`, `TwoFactorQuotientsContaining`. These can be used to obtain Ext information about a Weyl module.

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. Here the underlying Lie algebra is simply-connected and simple.

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). An example in Type D_4 is 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 modules.

2.1 Constructors

Here we consider functions that create Weyl modules, quotients of Weyl modules, submodules of Weyl modules, and submodules of quotient Weyl modules (subquotients).

2.1.1 WeylModule

▷ `WeylModule(p, wt, t, r)` (operation)
▷ `WeylModule(V, wt)` (operation)

Returns: a WeylModule

The first form of the command (with four arguments) constructs a Weyl module of highest weight `wt` in characteristic `p` of type `t` and rank `r`. The second form of the command (with two arguments) constructs a Weyl module of highest weight `wt` of the same characteristic and root system as the first parameter `V`, which must be an existing Weyl module.

Example

```
gap> V:= WeylModule(2,[1,0],"A",2);
V[ 1, 0 ]
gap> W:= WeylModule(V,[1,2]);
V[ 1, 2 ]
```

Weyl modules are attribute-storing objects (they remember their attributes after the first time they are computed).

2.1.2 QuotientWeylModule

▷ QuotientWeylModule(*S*) (operation)

Returns: a QuotientWeylModule

Constructs the quotient module V/S corresponding to the given submodule *S*. Here V is the ambient Weyl module of the given submodule *S*.

Example

```
gap> W:= WeylModule(2,[2,0],"A",2);
V[ 2, 0 ]
gap> Q:= QuotientWeylModule(SocleWeyl(W));
3-dimensional quotient of V[ 2, 0 ]
```

Quotient Weyl modules are attribute-storing objects.

2.1.3 SubWeylModule

▷ SubWeylModule(*V*, *v*) (operation)
▷ SubWeylModule(*S*, *v*) (operation)
▷ SubWeylModule(*V*, *list*) (operation)
▷ SubWeylModule(*S*, *list*) (operation)
▷ SubWeylModuleDirectSum(*V*, *list*) (operation)

Returns: a SubWeylModule

Submodules are attribute-storing objects. The first form constructs the submodule of the given Weyl module *V* generated by the given element *v*. The second form constructs the submodule (of the ambient Weyl module) generated by the given submodule *S* and given element *v*. The third form constructs the submodule of the given Weyl module *V* generated by the given element *list* of elements. The fourth form constructs the submodule generated by the given submodule *S* and given *list* of elements. The fifth form constructs the submodule of the Weyl module *V* which is the direct sum of the given list of submodules (which are assumed to be linearly independent, without checking).

Example

```
gap> W:= WeylModule(2,[2,0],"A",2);
V[ 2, 0 ]
gap> m:= MaximalVectors(W);
[ 1*v0, y1*v0 ]
gap> S:= SubWeylModule(W, m[2]);
3-dimensional submod of V[ 2, 0 ]
gap> SubWeylModule(S, m[1]);
6-dimensional submod of V[ 2, 0 ]
gap> SubWeylModule(W, m);
6-dimensional submod of V[ 2, 0 ]
gap> SubWeylModule(S, m);
6-dimensional submod of V[ 2, 0 ]
```

2.1.4 SubQuotientWeylModule

▷ SubWeylModule(*Q*, *v*) (operation)
▷ SubWeylModule(*S*, *v*) (operation)
▷ SubWeylModule(*Q*, *list*) (operation)
▷ SubWeylModule(*S*, *list*) (operation)

▷ SubWeylModuleDirectSum(*Q*, *list*) (operation)

Returns: a SubQuotientWeylModule

Subquotients are attribute-storing objects. The first form constructs the submodule of the given quotient Weyl module *Q* generated by the given element *v*. The second form constructs the subquotient Weyl module (of the ambient quotient) generated by the given subquotient *S* and given element *v*. The third form constructs the submodule of the given quotient Weyl module *Q* generated by the given element *list* of elements. The fourth form constructs the subquotient Weyl module generated by the given subquotient *S* and given *list* of elements. The fifth form constructs the submodule of the quotient Weyl module *Q* which is the direct sum of the given *list* of subquotients (which are assumed to be linearly independent, without checking).

Example

```
gap> W:= WeylModule(2,[2,0],"A",2);
V[ 2, 0 ]
gap> Q:= QuotientWeylModule(SocleWeyl(W));
3-dimensional quotient of V[ 2, 0 ]
gap> m:= MaximalVectors(Q);
[ 1*v0 ]
gap> S:= SubWeylModule(Q,m[1]);
3-dimensional submod of 3-dimensional quotient of V[ 2, 0 ]
gap> SubWeylModule(Q,m);
3-dimensional submod of 3-dimensional quotient of V[ 2, 0 ]
```

2.2 Filters

The following filters can be used to query whether or not an object has the indicated type.

2.2.1 IsWeylModule

▷ IsWeylModule(*V*) (filter)

Returns: true or false

2.2.2 IsQuotientWeylModule

▷ IsQuotientWeylModule(*V*) (filter)

Returns: true or false

2.2.3 IsSubWeylModule

▷ IsSubWeylModule(*V*) (filter)

Returns: true or false

2.2.4 IsSubQuotientWeylModule

▷ IsSubQuotientWeylModule(*V*) (filter)

Returns: true or false

2.3 Operations common to all four types

This section documents operations and attributes that can be applied to Weyl modules, quotients, submodules, and subquotients. In the following, `V` is a Weyl module, `Q` is a quotient, `S` is a submodule, and `T` is a subquotient. Section 2.1 documents the constructors for these objects. Furthermore, `wt` is a weight.

In all the examples in this section, we take `V`, `S`, `Q`, and `T` to be as follows.

Example

```
gap> V:=WeylModule(V,[4,0]);
V[ 4, 0 ]
gap> Dim(V);
15
gap> m:= MaximalVectors(V);
[ 1*v0, y1*v0 ]
gap> S:= SubWeylModule(V,m[2]);
9-dimensional submod of V[ 4, 0 ]
gap> Q:= QuotientWeylModule(S);
6-dimensional quotient of V[ 4, 0 ]
gap> mm:= MaximalVectors(Q);
[ 1*v0, y1^(2)*v0 ]
gap> T:= SubWeylModule(Q, mm[2]);
3-dimensional submod of 6-dimensional quotient of V[ 4, 0 ]
```

2.3.1 BasisVecs

▷ `BasisVecs(Q)` (operation)
▷ `BasisVecs(S)` (operation)
▷ `BasisVecs(T)` (operation)
▷ `BasisVecs(V)` (operation)

Example

```
gap> BasisVecs(V);
[ 1*v0, y1*v0, y3*v0, y1^(2)*v0, y1*y3*v0, y1^(3)*v0, y3^(2)*v0,
  y1^(2)*y3*v0, y1^(4)*v0, y1*y3^(2)*v0, y1^(3)*y3*v0, y3^(3)*v0,
  y1^(2)*y3^(2)*v0, y1*y3^(3)*v0, y3^(4)*v0 ]
gap> BasisVecs(S);
[ 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> BasisVecs(Q);
[ 1*v0, y1^(2)*v0, y3^(2)*v0, y1^(4)*v0, y1^(2)*y3^(2)*v0, y3^(4)*v0 ]
gap> BasisVecs(T);
[ y1^(2)*v0, y3^(2)*v0, y1^(2)*y3^(2)*v0 ]
```

2.3.2 Character

▷ `Character(Q)` (attribute)
▷ `Character(S)` (attribute)
▷ `Character(T)` (attribute)
▷ `Character(V)` (attribute)

Returns: a list

This function returns the character (as a list of weights and their multiplicities) of the given module.

Example

```
gap> Character(V);
[ [ 4, 0 ], 1, [ 2, 1 ], 1, [ 3, -1 ], 1, [ 0, 2 ], 1, [ 1, 0 ], 1,
  [ -2, 3 ], 1, [ 2, -2 ], 1, [ -1, 1 ], 1, [ -4, 4 ], 1, [ 0, -1 ], 1,
  [ -3, 2 ], 1, [ 1, -3 ], 1, [ -2, 0 ], 1, [ -1, -2 ], 1, [ 0, -4 ], 1 ]
gap> Character(Q);
[ [ 4, 0 ], 1, [ 0, 2 ], 1, [ 2, -2 ], 1, [ -4, 4 ], 1, [ -2, 0 ], 1,
  [ 0, -4 ], 1 ]
gap> Character(S);
[ [ 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 ]
gap> Character(T);
[ [ 0, 2 ], 1, [ 2, -2 ], 1, [ -2, 0 ], 1 ]
```

2.3.3 DecompositionNumbers

▷ DecompositionNumbers(*Q*) (attribute)
▷ DecompositionNumbers(*S*) (attribute)
▷ DecompositionNumbers(*T*) (attribute)
▷ DecompositionNumbers(*V*) (attribute)

Returns: a list

Returns a list of highest weights of simple composition factors (in the underlying characteristic) and their corresponding multiplicities in the given module.

Example

```
gap> DecompositionNumbers(V);
[ [ 4, 0 ], 1, [ 2, 1 ], 1, [ 0, 2 ], 1 ]
gap> DecompositionNumbers(Q);
[ [ 4, 0 ], 1, [ 0, 2 ], 1 ]
gap> DecompositionNumbers(S);
[ [ 2, 1 ], 1 ]
gap> DecompositionNumbers(T);
[ [ 0, 2 ], 1 ]
```

2.3.4 Dim

▷ Dim(*Q*) (attribute)
▷ Dim(*S*) (attribute)
▷ Dim(*T*) (attribute)
▷ Dim(*V*) (attribute)

Returns: an integer (the dimension of the module)

Example

```
gap> Dim(V);
15
gap> Dim(Q);
6
gap> Dim(S);
9
gap> Dim(T);
3
```

2.3.5 DominantWeights

▷ DominantWeights(*Q*) (attribute)
▷ DominantWeights(*S*) (attribute)
▷ DominantWeights(*T*) (attribute)
▷ DominantWeights(*V*) (attribute)

Returns: a list

This function lists the dominant weights occurring in the given module (with nonzero multiplicity). The multiplicities are not given.

Example

```
gap> DominantWeights(V);
[ [ 4, 0 ], [ 2, 1 ], [ 0, 2 ], [ 1, 0 ] ]
gap> DominantWeights(Q);
[ [ 4, 0 ], [ 0, 2 ] ]
gap> DominantWeights(S);
[ [ 2, 1 ], [ 1, 0 ] ]
gap> DominantWeights(T);
[ [ 0, 2 ] ]
```

2.3.6 DominantWeightSpaces

▷ DominantWeightSpaces(*Q*) (attribute)
▷ DominantWeightSpaces(*S*) (attribute)
▷ DominantWeightSpaces(*T*) (attribute)
▷ DominantWeightSpaces(*V*) (attribute)

Returns: a list

The output of this function is a list consisting of a weight followed by a list of basis vectors for the corresponding weight space.

Example

```
gap> DominantWeightSpaces(V);
[ [ 4, 0 ], [ 1*v0 ], [ 2, 1 ], [ y1*v0 ], [ 0, 2 ], [ y1^(2)*v0 ], [ 1, 0 ],
  [ y1*y3*v0 ] ]
gap> DominantWeightSpaces(Q);
[ [ 4, 0 ], [ 1*v0 ], [ 0, 2 ], [ y1^(2)*v0 ] ]
gap> DominantWeightSpaces(S);
[ [ 2, 1 ], [ y1*v0 ], [ 1, 0 ], [ y1*y3*v0 ] ]
gap> DominantWeightSpaces(T);
[ [ 0, 2 ], [ y1^(2)*v0 ] ]
```

2.3.7 Weights

▷ Weights(*Q*) (attribute)
▷ Weights(*S*) (attribute)
▷ Weights(*T*) (attribute)
▷ Weights(*V*) (attribute)

Returns: a list

Lists the weights in the given module.

Example

```
gap> Weights(V);
[ [ 4, 0 ], [ 2, 1 ], [ 3, -1 ], [ 0, 2 ], [ 1, 0 ], [ -2, 3 ], [ 2, -2 ],
```

```
    [ -1, 1 ], [ -4, 4 ], [ 0, -1 ], [ -3, 2 ], [ 1, -3 ], [ -2, 0 ],
    [ -1, -2 ], [ 0, -4 ] ]
gap> Weights(Q);
[ [ 4, 0 ], [ 0, 2 ], [ 2, -2 ], [ -4, 4 ], [ -2, 0 ], [ 0, -4 ] ]
gap> Weights(S);
[ [ 2, 1 ], [ 3, -1 ], [ 1, 0 ], [ -2, 3 ], [ 0, -1 ], [ -1, 1 ], [ 1, -3 ],
  [ -3, 2 ], [ -1, -2 ] ]
gap> Weights(T);
[ [ 0, 2 ], [ 2, -2 ], [ -2, 0 ] ]
```

2.3.8 WeightSpaces

▷ `WeightSpaces(`*Q*`)` (attribute)
▷ `WeightSpaces(`*S*`)` (attribute)
▷ `WeightSpaces(`*T*`)` (attribute)
▷ `WeightSpaces(`*V*`)` (attribute)

Returns: a list

Computes a list consisting of weights followed by a basis of their corresponding weight spaces, for each weight of the given module.

Example

```
gap> WeightSpaces(V);
[ [ 4, 0 ], [ 1*v0 ], [ 2, 1 ], [ y1*v0 ], [ 3, -1 ], [ y3*v0 ], [ 0, 2 ],
  [ y1^(2)*v0 ], [ 1, 0 ], [ y1*y3*v0 ], [ -2, 3 ], [ y1^(3)*v0 ], [ 2, -2 ],
  [ y3^(2)*v0 ], [ -1, 1 ], [ y1^(2)*y3*v0 ], [ -4, 4 ], [ y1^(4)*v0 ],
  [ 0, -1 ], [ y1*y3^(2)*v0 ], [ -3, 2 ], [ y1^(3)*y3*v0 ], [ 1, -3 ],
  [ y3^(3)*v0 ], [ -2, 0 ], [ y1^(2)*y3^(2)*v0 ], [ -1, -2 ],
  [ y1*y3^(3)*v0 ], [ 0, -4 ], [ y3^(4)*v0 ] ]
gap> WeightSpaces(Q);
[ [ 4, 0 ], [ 1*v0 ], [ 0, 2 ], [ y1^(2)*v0 ], [ 2, -2 ], [ y3^(2)*v0 ],
  [ -4, 4 ], [ y1^(4)*v0 ], [ -2, 0 ], [ y1^(2)*y3^(2)*v0 ], [ 0, -4 ],
  [ y3^(4)*v0 ] ]
gap> WeightSpaces(S);
[ [ 2, 1 ], [ y1*v0 ], [ 3, -1 ], [ y3*v0 ], [ 1, 0 ], [ y1*y3*v0 ],
  [ -2, 3 ], [ y1^(3)*v0 ], [ 0, -1 ], [ y1*y3^(2)*v0 ], [ -1, 1 ],
  [ y1^(2)*y3*v0 ], [ 1, -3 ], [ y3^(3)*v0 ], [ -3, 2 ], [ y1^(3)*y3*v0 ],
  [ -1, -2 ], [ y1*y3^(3)*v0 ] ]
gap> WeightSpaces(T);
[ [ 0, 2 ], [ y1^(2)*v0 ], [ 2, -2 ], [ y3^(2)*v0 ], [ -2, 0 ],
  [ y1^(2)*y3^(2)*v0 ] ]
```

2.3.9 WeightSpace

▷ `WeightSpace(`*Q*`, `*wt*`)` (operation)
▷ `WeightSpace(`*S*`, `*wt*`)` (operation)
▷ `WeightSpace(`*T*`, `*wt*`)` (operation)
▷ `WeightSpace(`*V*`, `*wt*`)` (operation)

Returns: a list

Gives a basis for the weight space of the given weight *wt*.

Example

```
gap> WeightSpace(V, [2,1]);
[ y1*v0 ]
gap> WeightSpace(Q, [2,1]);
[  ]
gap> WeightSpace(S, [2,1]);
[ y1*v0 ]
gap> WeightSpace(T, [2,-2]);
[ y3^(2)*v0 ]
```

2.3.10 TheCharacteristic

▷ TheCharacteristic(*Q*) (operation)
▷ TheCharacteristic(*S*) (operation)
▷ TheCharacteristic(*T*) (operation)
▷ TheCharacteristic(*V*) (operation)

Returns: an integer

Gives the characteristic of the base field.

Example

```
gap> TheCharacteristic(V);
2
gap> TheCharacteristic(Q);
2
gap> TheCharacteristic(S);
2
gap> TheCharacteristic(T);
2
```

2.3.11 TheLieAlgebra

▷ TheLieAlgebra(*Q*) (attribute)
▷ TheLieAlgebra(*S*) (attribute)
▷ TheLieAlgebra(*T*) (attribute)
▷ TheLieAlgebra(*V*) (attribute)

Returns: a Lie algebra

Gives the underlying Lie algebra for the given module.

Example

```
gap> TheLieAlgebra(V);
<Lie algebra of dimension 8 over Rationals>
gap> TheLieAlgebra(Q);
<Lie algebra of dimension 8 over Rationals>
gap> TheLieAlgebra(S);
<Lie algebra of dimension 8 over Rationals>
gap> TheLieAlgebra(T);
<Lie algebra of dimension 8 over Rationals>
```

The GAP manual gives additional operations for various properties and attributes of such Lie algebras and their enveloping algebras.

2.4 Operations common to Weyl modules, submodules, and quotients

In the following, `Q` is a quotient Weyl module, `S` is a submodule, and `V` is a Weyl module.

2.4.1 MaximalVectors

▷ `MaximalVectors(Q, wt)` (operation)
▷ `MaximalVectors(Q)` (attribute)
▷ `MaximalVectors(S, wt)` (operation)
▷ `MaximalVectors(S)` (attribute)
▷ `MaximalVectors(V, wt)` (operation)
▷ `MaximalVectors(V)` (attribute)

Returns: a list

Returns a list of linearly independent maximal vectors for the given dominant weight `wt` or for all dominant weights of the module. The maximal vectors of a particular weight form a basis of the space of maximal vectors of that weight.

In the following example, we assume that `V`, `Q`, and `S` are the same modules as defined in the example at the beginning of Section 2.3.

Example

```
gap> MaximalVectors(V);
[ 1*v0, y1*v0 ]
gap> m:= MaximalVectors(V);
[ 1*v0, y1*v0 ]
gap> List(m, Weight);
[ [ 4, 0 ], [ 2, 1 ] ]
gap> MaximalVectors(V, [2,1]);
[ y1*v0 ]
gap> m:= MaximalVectors(Q);
[ 1*v0, y1^(2)*v0 ]
gap> List(m, Weight);
[ [ 4, 0 ], [ 0, 2 ] ]
gap> MaximalVectors(Q, [0,2]);
[ y1^(2)*v0 ]
gap> m:= MaximalVectors(S);
[ y1*v0 ]
gap> List(m, Weight);
[ [ 2, 1 ] ]
gap> MaximalVectors(S, [2,1]);
[ y1*v0 ]
```

2.4.2 IsAmbiguous

▷ `IsAmbiguous(Q)` (attribute)
▷ `IsAmbiguous(S)` (attribute)
▷ `IsAmbiguous(V)` (attribute)

Returns: true or false

The module is ambiguous if it has two or more linearly independent maximal vectors of the same weight.

Example

```
gap> V:= WeylModule(2,[3,0],"G",2);
V[ 3, 0 ]
gap> IsAmbiguous(V);
true
```

2.4.3 AmbiguousMaxVecs

▷ AmbiguousMaxVecs(*Q*) (attribute)
▷ AmbiguousMaxVecs(*S*) (attribute)
▷ AmbiguousMaxVecs(*V*) (attribute)

Returns: a list

This function lists a basis for the subspace of ambiguous maximal vectors.

Example

```
gap> V:= WeylModule(2,[3,0],"G",2);
V[ 3, 0 ]
gap> AmbiguousMaxVecs(V);
[ y1*y3*v0, y4*v0 ]
```

2.5 Operations on Weyl modules and their quotients

In the following, *Q* is a quotient Weyl module and *V* is a Weyl module.

2.5.1 ActOn

▷ ActOn(*Q*, *u*, *v*) (operation)
▷ ActOn(*V*, *u*, *v*) (operation)

Returns: An element of the Weyl module or quotient

This function returns the result of acting by the hyperalgebra element *u* on the given vector *v*. Here *v* must be an element of the given Weyl module *V* or quotient Weyl module *Q*. The command LatticeGeneratorsInUEA is a pre-existing GAP command; see the chapter on Lie algebras in the GAP reference manual for further details. The lattice generators are regarded as standard generators of the hyperalgebra for computing the action.

Example

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

2.5.2 Generator

▷ Generator(*Q*) (operation)
▷ Generator(*V*) (operation)

Returns: a highest weight vector

This returns a generating vector of the given module.

Example

```
gap> V:= WeylModule(2,[3,0],"G",2);
V[ 3, 0 ]
gap> Generator(V);
1*v0
```

2.5.3 GensSocleLayers

▷ GensSocleLayers(*Q*) (attribute)
▷ GensSocleLayers(*V*) (attribute)

Returns: a list

Returns a list of lists, such that the *i*th list gives a list of generators of the *i*th module in the socle series of the input.

Example

```
gap> V:= WeylModule(2,[3,0],"G",2);
V[ 3, 0 ]
gap> GensSocleLayers(V);
[ [ y1*y4*v0, y1*y3*y4*v0+y1*y6*v0+y3*y5*v0,
      y1*y4*y6*v0+y3*y4*y5*v0+y4^(3)*v0 ], [ y4*v0, y1^(2)*y3*y6*v0 ],
  [ y1*y3*v0 ], [ y5*y6*v0 ], [ y1*y6*v0+y4^(2)*v0 ], [ 1*v0 ] ]
```

2.5.4 PrintSocleLayers

▷ PrintSocleLayers(*Q*) (operation)
▷ PrintSocleLayers(*V*) (operation)

Returns: nothing

Prints the weights of the generators of the socle layers corresponding to the output of the previous command. These are the highest weights of the simple composition factors in each socle layer.

In the following example, we assume that *V* is the same Weyl module defined in the preceding example.

Example

```
gap> PrintSocleLayers(V);
Printing highest weights of simples in socle layers of V[ 3, 0 ]
Layer 1: [ [ 0, 1 ], [ 1, 0 ], [ 0, 0 ] ]
Layer 2: [ [ 2, 0 ], [ 0, 0 ] ]
Layer 3: [ [ 2, 0 ] ]
Layer 4: [ [ 0, 0 ] ]
Layer 5: [ [ 1, 0 ] ]
Layer 6: [ [ 3, 0 ] ]
```

2.5.5 SocleSeries

▷ SocleSeries(*Q*) (attribute)
▷ SocleSeries(*V*) (attribute)

Returns: a list

Returns the socle series of its input module, as a list of submodules.

Example

```
gap> V:= WeylModule(2,[3,0],"G",2);
V[ 3, 0 ]
gap> SocleSeries(V);
[ 21-dimensional submod of V[ 3, 0 ], 28-dimensional submod of V[ 3, 0 ],
  34-dimensional submod of V[ 3, 0 ], 35-dimensional submod of V[ 3, 0 ],
  41-dimensional submod of V[ 3, 0 ], 77-dimensional submod of V[ 3, 0 ] ]
```

2.5.6 SocleWeyl

▷ SocleWeyl(*V*) (attribute)
▷ SocleWeyl(*Q*) (attribute)

Returns: a submodule

This function returns the socle of the given module.

Example

```
gap> V:= WeylModule(2,[3,0],"G",2);
V[ 3, 0 ]
gap> SocleWeyl(V);
21-dimensional submod of V[ 3, 0 ]
```

2.6 Operations on Weyl modules

This section documents additional operations that can be applied to an existing Weyl module *V*.

2.6.1 MaximalSubmodule

▷ MaximalSubmodule(*V*) (attribute)

Returns: a submodule

Calculates and returns the unique maximal submodule of the given Weyl module *V*.

Example

```
gap> V:= WeylModule(3,[3,0],"A",2);
V[ 3, 0 ]
gap> MaximalSubmodule(V);
7-dimensional submod of V[ 3, 0 ]
```

2.6.2 SimpleQuotient

▷ SimpleQuotient(*V*) (attribute)

Returns: a quotient Weyl module

Calculates and returns the simple quotient by the unique maximal submodule of the given Weyl module.

Example

```
gap> V:= WeylModule(3,[3,0],"A",2);
V[ 3, 0 ]
gap> Q:= SimpleQuotient(V);
3-dimensional quotient of V[ 3, 0 ]
```

2.7 Operations on quotients

In the following, `Q` is a quotient Weyl module.

2.7.1 AmbientWeylModule

▷ `AmbientWeylModule(Q)` (operation)

Returns: a Weyl module

This returns the ambient Weyl module `V` corresponding to the given quotient `Q`.

In the following example, we assume that `Q` is the same as the quotient module defined in the preceding example.

Example

```
gap> AmbientWeylModule(Q);
V[ 3, 0 ]
```

2.7.2 DefiningKernel

▷ `DefiningKernel(Q)` (operation)

Returns: a submodule

This returns the kernel corresponding to the given quotient `Q`. In other words, it returns the submodule `S` such that Q is isomorphic to V/S, where `V` is the ambient Weyl module.

In the following example, we assume that `Q` is the same as the quotient module in the preceding example.

Example

```
gap> DefiningKernel(Q);
7-dimensional submod of V[ 3, 0 ]
```

2.8 Operations on submodules

In the following, `S` is a submodule.

2.8.1 AmbientWeylModule

▷ `AmbientWeylModule(S)` (operation)

Returns: a Weyl module

This function returns the ambient Weyl module containing the given submodule `S`.

Example

```
gap> V:= WeylModule(3,[3,0],"A",2);
V[ 3, 0 ]
gap> S:= MaximalSubmodule(V);
7-dimensional submod of V[ 3, 0 ]
```

```
gap> AmbientWeylModule(S);
V[ 3, 0 ]
```

2.8.2 Generators

▷ `Generators(S)` (operation)

Returns: a list

Returns a list of generators of the given submodule S. In the following example, we assume that S is the same as in the preceding example.

Example

```
gap> Generators(S);
[ y1*v0 ]
```

2.8.3 IsWithin

▷ `IsWithin(S, v)` (operation)

Returns: true or false

This function returns `true` if and only if the given vector v lies in the given submodule S.

Example

```
gap> V:= WeylModule(3,[3,0],"A",2);
V[ 3, 0 ]
gap> S:= MaximalSubmodule(V);
7-dimensional submod of V[ 3, 0 ]
gap> g:= Generators(S);
[ y1*v0 ]
gap> IsWithin(S, g[1]);
true
```

2.8.4 NextSocle

▷ `NextSocle(S)` (operation)

Returns: a SubWeylModule

This function returns the maximal submodule T containing the given submodule S such that T/S is semisimple. If S happens to be an element of the socle series then the function returns the next element in the socle series.

Example

```
gap> W:= WeylModule(2,[3,0],"G",2);
V[ 3, 0 ]
gap> g:= Generators(SocleWeyl(W));
[ y1*y4*v0, y1*y3*y4*v0+y1*y6*v0+y3*y5*v0, y1*y4*y6*v0+y3*y4*y5*v0+y4^(3)*v0 ]
gap> S:= SubWeylModule(W, g[1]);
14-dimensional submod of V[ 3, 0 ]
gap> T:= NextSocle(S);
21-dimensional submod of V[ 3, 0 ]
gap> DecompositionNumbers(T);
[ [ 0, 1 ], 1, [ 1, 0 ], 1, [ 0, 0 ], 1 ]
```

In the above example, S is a simple submodule of the socle, and `NextSocle(S)` computes an extension of it by two simples.

2.8.5 GensNextSocle

▷ GensNextSocle(*S*) (operation)

Returns: a list

This function returns a list of generators of the submodule returned by NextSocle(S). In the example below, *S* is the submodule constructed in the preceding example.

Example

```
gap> g:= GensNextSocle(S);
[ y1*y6*v0+y3*y5*v0, y4^(3)*v0 ]
gap> List(g, Weight);
[ [ 1, 0 ], [ 0, 0 ] ]
```

2.8.6 SocleLengthTwoQuotient

▷ SocleLengthTwoQuotient(*S*) (attribute)

Returns: a QuotientWeylModule

This function returns a quotient of the ambient Weyl module *V* with socle series length at most two such that *S* lies in its defining kernel.

Example

```
gap> W:= WeylModule(2,[3,0],"G",2);
V[ 3, 0 ]
gap> ss:= SocleSeries(W);
[ 21-dimensional submod of V[ 3, 0 ], 28-dimensional submod of V[ 3, 0 ],
  34-dimensional submod of V[ 3, 0 ], 35-dimensional submod of V[ 3, 0 ],
  41-dimensional submod of V[ 3, 0 ], 77-dimensional submod of V[ 3, 0 ] ]
gap> amv:= AmbiguousMaxVecs(W);
[ y1*y3*v0, y4*v0 ]
gap> Q1:= SocleLengthTwoQuotient(ss[4]);
42-dimensional quotient of V[ 3, 0 ]
gap> PrintSocleLayers(Q1);
Printing highest weights of simples in socle layers of
42-dimensional quotient of V[ 3, 0 ]
Layer 1: [ [ 1, 0 ] ]
Layer 2: [ [ 3, 0 ] ]
gap> Q2:= SocleLengthTwoQuotient(SubWeylModule(W,amv[1]+amv[2]));
48-dimensional quotient of V[ 3, 0 ]
gap> PrintSocleLayers(Q2);
Printing highest weights of simples in socle layers of
48-dimensional quotient of V[ 3, 0 ]
Layer 1: [ [ 2, 0 ], [ 1, 0 ] ]
Layer 2: [ [ 3, 0 ] ]
```

Here we see an example of an ambiguous Weyl module with different quotients of socle length two.

2.8.7 TwoFactorQuotientsContaining

▷ TwoFactorQuotientsContaining(*S*) (attribute)

Returns: a list of QuotientWeylModules

This returns a list of quotients of the ambient Weyl module, each having exactly two composition factors, each of which contain *S* in their defining kernel. Such quotients realize non-split extensions

of the simple top composition factor of V. NOTE. Even when S is the trivial module, we do not claim that the output will give *all* of the extensions.

In the following example, we assume that W, amv are as defined in the preceding example.

Example

```
gap> Q:= TwoFactorQuotientsContaining(SubWeylModule(W,amv[1]+amv[2]));
[ 42-dimensional quotient of V[ 3, 0 ], 42-dimensional quotient of V[ 3, 0 ] ]
gap> PrintSocleLayers(Q[1]);
Printing highest weights of simples in socle layers of
42-dimensional quotient of V[ 3, 0 ]
Layer 1: [ [ 1, 0 ] ]
Layer 2: [ [ 3, 0 ] ]
gap> PrintSocleLayers(Q[2]);
Printing highest weights of simples in socle layers of
42-dimensional quotient of V[ 3, 0 ]
Layer 1: [ [ 2, 0 ] ]
Layer 2: [ [ 3, 0 ] ]
```

Here we see that the ambient Weyl module has at least two non-isomorphic extensions realized in its second radical. Comparing with information from an earlier example (see PrintSocleLayers (2.5.4)) reveals that the Weyl module in question is non-rigid (its socle and radical series do not coincide).

2.9 Operations on subquotients

In the following, T is a subquotient.

2.9.1 AmbientQuotient

▷ AmbientQuotient(T) (operation)

Returns: a QuotientWeylModule

This function returns the ambient quotient Weyl module containing the given subquotient T.

Example

```
gap> W:= WeylModule(2,[3,0],"G",2);
V[ 3, 0 ]
gap> m:= AmbiguousMaxVecs(W); List(m, Weight);
[ y1*y3*v0, y4*v0 ]
[ [ 2, 0 ], [ 2, 0 ] ]
gap> Q:= QuotientWeylModule(SubWeylModule(W,m[1]));
64-dimensional quotient of V[ 3, 0 ]
gap> subQ:= SubWeylModule(Q, m[2]);
21-dimensional submod of 64-dimensional quotient of V[ 3, 0 ]
gap> AmbientQuotient(subQ);
64-dimensional quotient of V[ 3, 0 ]
```

2.9.2 Generators

▷ Generators(T) (operation)

Returns: a list

This returns a list of generators for the given subquotient T. In the next example, we assume that subQ is the subquotient constructed in the example for the AmbientQuotient (2.9.1) command, documented above.

Example

```
gap> Generators(subQ);
[ y4*v0 ]
```

2.9.3 IsWithin

▷ IsWithin(T, v) (operation)

Returns: true or false

This returns true if and only if the image of the given vector v (under the quotient map from the ambient Weyl module to the ambient quotient) lies in the given subquotient T. In the next example, we assume that subQ, Q, and m are as defined in the example for AmbientQuotient (2.9.1) above.

Example

```
gap> IsWithin(subQ,m[2]);
true
gap> IsWithin(subQ, Generator(Q));
false
```

2.9.4 NextSocle

▷ NextSocle(T) (operation)

Returns: a SubQuotientWeylModule

This function returns the maximal subquotient T containing the given subquotient S such that T/S is semisimple. If S happens to be an element of the socle series then the function returns the next element in the socle series.

In the next example, we assume that subQ is the subquotient constructed in the example for the AmbientQuotient (2.9.1) command, documented above.

Example

```
gap> DecompositionNumbers(subQ);
[ [ 2, 0 ], 1, [ 0, 1 ], 1, [ 0, 0 ], 1 ]
gap> N:= NextSocle(subQ);
22-dimensional submod of 64-dimensional quotient of V[ 3, 0 ]
gap> DecompositionNumbers(N);
[ [ 2, 0 ], 1, [ 0, 1 ], 1, [ 0, 0 ], 2 ]
```

Chapter 3

Weights and Characters

This chapter documents additional functions available for computation of weights and characters.

3.1 Weights

3.1.1 Weight (for IsLeftAlgebraModuleElement)

▷ `Weight(elt)` (operation)

Returns: a list of integers

The weight of the given element *elt* is calculated and returned.

Example

```
gap> V:= WeylModule(3,[3,3],"A",2);
V[ 3, 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> Weight(m[2]);
[ 1, 4 ]
gap> List(m,Weight);
[ [ 3, 3 ], [ 1, 4 ], [ 4, 1 ], [ 0, 3 ], [ 3, 0 ], [ 1, 1 ] ]
```

NOTE. The above trick of applying the `Weight` function across an entire list *lst* of vectors, with the command `List(lst, Weight)`, is very useful in many situations. This capability is built in to the `List` function in GAP.

3.2 Characters

We have already seen the function `Character` (2.3.2), that computes the (formal) character of a given Weyl module, quotient, submodule, or subquotient. We now consider some additional functions for computing characters.

3.2.1 DecomposeCharacter

▷ `DecomposeCharacter(ch, p, typ, rk)` (operation)

Returns: a list (of simple highest weights and their multiplicities)

If *ch* is a given character (of some module) then this function computes the multiplicities of the simple characters in *ch*, thus obtaining the decomposition numbers of the module. Here it is necessary to specify the characteristic *p* and root system (of type *typ* and rank *rk*) for the simple characters. For instance, this can be used to decompose tensor products.

Example

```
gap> V:= WeylModule(2,[2,0],"A",2);
V[ 2, 0 ]
gap> ch:= ProductCharacter(Character(V),Character(V));
[ [ 4, 0 ], 1, [ 2, 1 ], 2, [ 3, -1 ], 2, [ 0, 2 ], 3, [ 1, 0 ], 4,
  [ 2, -2 ], 3, [ -2, 3 ], 2, [ -1, 1 ], 4, [ 0, -1 ], 4, [ 1, -3 ], 2,
  [ -4, 4 ], 1, [ -3, 2 ], 2, [ -2, 0 ], 3, [ -1, -2 ], 2, [ 0, -4 ], 1 ]
gap> DecomposeCharacter(ch,2,"A",2);
[ [ 4, 0 ], 1, [ 2, 1 ], 2, [ 0, 2 ], 3, [ 1, 0 ], 2 ]
```

3.2.2 DifferenceCharacter

▷ `DifferenceCharacter(`*ch1, ch2*`)` (operation)

Returns: a list (a character)

If *ch1* and *ch2* are given characters, this function returns their formal difference character. It is used in the definition of the `DecomposeCharacter` function.

Example

```
gap> DifferenceCharacter(Character(V),Character(V));
[  ]
```

The empty list here implements the zero character.

3.2.3 ProductCharacter

▷ `ProductCharacter(`*ch1, ch2*`)` (operation)

Returns: a list (a character)

Returns the product character of its inputs *ch1* and *ch2*. If *ch1* and *ch2* are characters of modules then the output of this function is the character of the tensor product of the modules.

Example

```
gap> V:= WeylModule(2,[2,0],"A",2);
V[ 2, 0 ]
gap> ch:= ProductCharacter(Character(V),Character(V));
[ [ 4, 0 ], 1, [ 2, 1 ], 2, [ 3, -1 ], 2, [ 0, 2 ], 3, [ 1, 0 ], 4,
  [ 2, -2 ], 3, [ -2, 3 ], 2, [ -1, 1 ], 4, [ 0, -1 ], 4, [ 1, -3 ], 2,
  [ -4, 4 ], 1, [ -3, 2 ], 2, [ -2, 0 ], 3, [ -1, -2 ], 2, [ 0, -4 ], 1 ]
```

By applying the function `DecomposeCharacter` (3.2.1) we can decompose tensor products in positive characteristic.

3.2.4 SimpleCharacter

▷ `SimpleCharacter(`*p, wt, typ, rk*`)` (operation)

Returns: a list (a character)

Computes the simple character of highest weight *wt* in characteristic *p*. The arguments *typ* and *rk* specify the type and rank of the underlying root system. The function uses Steinberg's tensor product theorem.

Example

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

Another way to compute the same result is to compute the `Character` of the output of `SimpleQuotient(V)`, where `V` is the WeylModule in the same characteristic and root system with the same highest weight.

Example

```
gap> V:= WeylModule(2,[2,0],"A",2);
V[ 2, 0 ]
gap> Character(SimpleQuotient(V));
[ [ 2, 0 ], 1, [ -2, 2 ], 1, [ 0, -2 ], 1 ]
```

Chapter 4

Schur Algebras

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.

NOTE. The `SymmetricGroupDecompositionMatrix` (4.2.3) function for symmetric group decomposition numbers is quite slow, so readers interested in symmetric group computations may want to look elsewhere for more effcient tools.

4.1 Constructor and filter

Weyl modules for a Schur algebra are constructed by the following.

4.1.1 SchurAlgebraWeylModule

▷ `SchurAlgebraWeylModule(p, ptn)` (operation)

Returns: a Weyl module

This function creates and returns a Weyl module of highest weight defined by the given partition `ptn`. The length of the partition, which may be padded by zeros as necessary, defines the underlying GL_n and the Schur algebra degree.

Example

```
gap> V:= SchurAlgebraWeylModule(3,[1,1,0]);
Schur algebra module V[ 1, 1, 0 ]
```

Here we define the Weyl module for GL_3 of highest weight [1, 1] in the partition notation.

4.1.2 IsSchurAlgebraWeylModule

▷ `IsSchurAlgebraWeylModule(V)` (filter)

Returns: true or false

Returns true if and only if the given `V` is a Schur algebra Weyl module.

4.2 Decomposition matrices

Decomposition matrices for Schur algebras and symmteric groups in positive characteristic can be computed.

4.2.1 SchurAlgebraDecompositionMatrix

▷ `SchurAlgebraDecompositionMatrix(p, n, r)` (operation)

Returns: a matrix

Returns the decomposition matrix for the Schur algebra $S(n,r)$ in characteristc p. The rows and columns of the matrix are indexed by the partitions produced by `BoundedPartitions(n,r)` ordered the same as in the output of that function.

Example

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

Here we compute the decomposition matrix for $S(4,3)$ in characteristic 3. The rows and columns of the matrix are indexed by the following partitions:

Example

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

4.2.2 SymmetricGroupDecompositionNumbers

▷ `SymmetricGroupDecompositionNumbers(p, ptn)` (operation)

Returns: a list

Returns the decomposition numbers of the dual Specht module indexed by the given partition *ptn* in characteristic *p*.

Example

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

4.2.3 SymmetricGroupDecompositionMatrix

▷ `SymmetricGroupDecompositionMatrix(p, n)` (operation)

Returns: a matrix

Returns the decomposition matrix for the symmetric group on *n* letters in characteristc *p*. The rows of the matrix are labeled by the partitions of *n* in the order produced by `AllPartitions(n)`, and the columns are labeled by the *p*-restricted partitions of *n*. NOTE. GAP has a built-in `Partitions` function that also gives all the partitions of *n*, but the ordering is different.

Example

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

4.3 Partitions

This section documents a number of functions for converting between weights and partitions (in type A) as well as other related functions.

4.3.1 CompositionToWeight

▷ `CompositionToWeight(mu)` (operation)

Returns: a list (a weight)

This converts the given composition *mu* into a weight by taking successive differences of its parts.

Example

```
gap> CompositionToWeight([1,2,0,1]);
[ -1, 2, -1 ]
```

4.3.2 WeightToComposition

▷ `WeightToComposition(r, wt)` (operation)

Returns: a list (a composition) or `fail`

This converts the given weight *wt* into a composition of degree *r*. Without degree information, this function is ill defined. Returns `fail` if the operation is impossible.

Example

```
gap> WeightToComposition(4,[-1, 2, -1]);
[ 1, 2, 0, 1 ]
gap> WeightToComposition(8,[-1, 2, -1]);
[ 2, 3, 1, 2 ]
gap> WeightToComposition(6,[-1, 2, -1]);
fail
```

4.3.3 AllPartitions

▷ `AllPartitions(n)` (operation)

Returns: a list of partitions

Lists all the partitions of *n*. Note that GAP has a built-in Partitions function that also gives all the partitions of *n*, but with a different ordering.

Example

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

4.3.4 BoundedPartitions

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

▷ `BoundedPartitions(n, r)` (operation)

Returns: a list of partitions

Returns a list of *n* part partitions of degree *r* such that each part lies in the closed interval [0,*s*]. The second form returns a list of *n* part partitions of degree *r*. Note that `BoundedPartitions(n,r)` is equivalent to `BoundedPartitions(n,r,r)`.

Example

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

4.3.5 Conjugate

▷ `Conjugate(ptn)` (operation)

Returns: a list (a partition)

Returns the conjugate partition of *ptn*.

Example

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

4.3.6 pRegular

▷ `pRegular(p, ptn)` (operation)

Returns: true or false

Returns true if and only if the given partition *ptn* is *p*-regular.

Example

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

4.3.7 pRegularPartitions

▷ `pRegularPartitions(p, n)` (operation)

Returns: a list

Returns a list of all *p*-regular partitions of *n*.

Example

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

4.3.8 pRestricted

▷ `pRestricted(p, ptn)` (operation)

Returns: true or false

Returns true if and only if the given partition *ptn* is *p*-restricted.

Example

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

4.3.9 pRestrictedPartitions

▷ pRestrictedPartitions(*p*, *n*) (operation)

Returns: a list

Returns a list of all *p*-restricted partitions of *n*.

Example

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

4.3.10 Mullineux

▷ Mullineux(*p*, *mu*) (operation)

Returns: a list

Applies the Mullineux map to the partition *mu* in characteristic *p*.

Example

```
gap> Mullineux(2,[1,1,1]);
p-singular!
gap> Mullineux(2,[3]);
[ 3 ]
gap> Mullineux(3,[3]);
[ 2, 1 ]
gap> Mullineux(3,[2,1]);
[ 3 ]
```

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

[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

[Irv86] Ronald S. Irving. The structure of certain highest weight modules for SL_3. *J. Algebra*, 99(2):438–457, 1986. 4

[Jan03] Jens Carsten Jantzen. *Representations of algebraic groups*, volume 107 of *Mathematical Surveys and Monographs*. American Mathematical Society, Providence, RI, second edition, 2003. 4

[Xi99] Nanhua Xi. Maximal and primitive elements in Weyl modules for type A_2. *J. Algebra*, 215(2):735–756, 1999. 4

Index