Handbook of Magma Functions

Volume 8

Lie Theory

John Cannon Wieb Bosma
Claus Fieker Allan Steel
Editors

Version 2.22
Sydney
June 9, 2016

MAGMA
COMPUTER ALGEBRA

Handbook of Magma Functions

Editors:

John Cannon *Wieb Bosma* *Claus Fieker* *Allan Steel*

Handbook Contributors:

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

Production Editors:

Wieb Bosma *Claus Fieker* *Allan Steel* *Nicole Sutherland*

HTML Production:

Claus Fieker *Allan Steel*

VOLUME 8: OVERVIEW

XIII LIE THEORY 3031

100 INTRODUCTION TO LIE THEORY 3033
101 COXETER SYSTEMS 3039
102 ROOT SYSTEMS 3063
103 ROOT DATA 3085
104 COXETER GROUPS 3137
105 REFLECTION GROUPS 3177
106 LIE ALGEBRAS 3209
107 KAC-MOODY LIE ALGEBRAS 3297
108 QUANTUM GROUPS 3307
109 GROUPS OF LIE TYPE 3333
110 REPRESENTATIONS OF LIE GROUPS AND ALGEBRAS 3373

VOLUME 8: CONTENTS

XIII LIE THEORY 3031

100 INTRODUCTION TO LIE THEORY 3033

100.1 Descriptions of Coxeter Groups 3035
100.2 Root Systems and Root Data 3036
100.3 Coxeter and Reflection Groups 3036
100.4 Lie Algebras and Groups of Lie Type 3037
100.5 Highest Weight Representations 3037
100.6 Universal Enveloping Algebras and Quantum Groups 3037
100.7 Bibliography 3038

101 COXETER SYSTEMS 3039

101.1 Introduction 3041
101.2 Coxeter Matrices 3041
101.3 Coxeter Graphs 3043
101.4 Cartan Matrices 3045
101.5 Dynkin Digraphs 3048
101.6 Finite and Affine Coxeter Groups 3050
101.7 Hyperbolic Groups 3058
101.8 Related Structures 3059
101.9 Bibliography 3061

102 ROOT SYSTEMS 3063

102.1 Introduction 3065
102.1.1 Reflections 3065
102.1.2 Definition of a Root System 3065
102.1.3 Simple and Positive Roots 3066
102.1.4 The Coxeter Group 3066
102.1.5 Nonreduced Root Systems 3067
102.2 Constructing Root Systems 3067
102.3 Operators on Root Systems 3071
102.4 Properties of Root Systems 3073
102.5 Roots and Coroots 3074
102.5.1 Accessing Roots and Coroots 3074
102.5.2 Reflections 3077
102.5.3 Operations and Properties for Roots and Coroot Indices 3079
102.6 Building Root Systems 3082
102.7 Related Structures 3084
102.8 Bibliography 3084

103 ROOT DATA 3085

103.1 Introduction *3089*
103.1.1 Reflections 3089
103.1.2 Definition of a Split Root Datum 3090
103.1.3 Simple and Positive Roots 3090
103.1.4 The Coxeter Group 3090
103.1.5 Nonreduced Root Data 3091
103.1.6 Isogeny of Split Reduced Root Data 3091
103.1.7 Extended Root Data 3092
103.2 Constructing Root Data *3092*
103.2.1 Constructing Sparse Root Data 3098
103.3 Operations on Root Data *3100*
103.4 Properties of Root Data *3107*
103.5 Roots, Coroots and Weights *3110*
103.5.1 Accessing Roots and Coroots 3110
103.5.2 Reflections 3117
103.5.3 Operations and Properties for Root and Coroot Indices 3119
103.5.4 Weights 3122
103.6 Building Root Data *3124*
103.7 Morphisms of Root Data *3130*
103.8 Constants Associated with Root Data *3132*
103.9 Related Structures *3134*
103.10 Bibliography *3135*

104 COXETER GROUPS 3137

104.1 Introduction *3139*
104.1.1 The Normal Form for Words 3140
104.2 Constructing Coxeter Groups *3140*
104.3 Converting Between Types of Coxeter Group *3143*
104.4 Operations on Coxeter Groups *3146*
104.5 Properties of Coxeter Groups *3150*
104.6 Operations on Elements *3152*
104.7 Roots, Coroots and Reflections *3154*
104.7.1 Accessing Roots and Coroots 3154
104.7.2 Operations and Properties for Root and Coroot Indices 3157
104.7.3 Weights 3159
104.8 Reflections *3160*
104.9 Reflection Subgroups *3162*
104.10 Root Actions *3166*
104.11 Standard Action *3167*
104.12 Braid Groups *3168*
104.13 W-graphs *3169*
104.14 Related Structures *3174*
104.15 Bibliography *3174*

105 REFLECTION GROUPS 3177

105.1 Introduction *3179*
105.2 Construction of Pseudo-reflections *3179*
105.2.1 Pseudo-reflections Preserving Reflexive Forms 3182
105.3 Construction of Reflection Groups *3184*
105.4 Construction of Real Reflection Groups *3185*
105.5 Construction of Finite Complex Reflection Groups *3188*
105.6 Operations on Reflection Groups *3196*
105.7 Properties of Reflection Groups *3200*
105.8 Roots, Coroots and Reflections *3202*
105.8.1 Accessing Roots and Coroots 3202
105.8.2 Reflections 3205
105.8.3 Weights 3206
105.9 Related Structures *3208*
105.10 Bibliography *3208*

106 LIE ALGEBRAS 3209

106.1 Introduction *3213*
106.1.1 Guide for the Reader 3213
106.2 Constructors for Lie Algebras *3214*
106.3 Finitely Presented Lie Algebras *3217*
106.3.1 Construction of the Free Lie Algebra 3218
106.3.2 Properties of the Free Lie Algebra 3218
106.3.3 Operations on Elements of the Free Lie Algebra 3219
106.3.4 Construction of a Finitely-Presented Lie Algebra 3220
106.3.5 Homomorphisms of the Free Lie Algebra 3224
106.4 Lie Algebras Generated by Extremal Elements *3225*
106.4.1 Constructing Lie Algebras Generated by Extremal Elements 3226
106.4.2 Properties of Lie Algebras Generated by Extremal Elements 3227
106.4.3 Instances of Lie Algebras Generated by Extremal Elements 3231
106.4.4 Studying the Parameter Space 3233
106.5 Families of Lie Algebras *3236*
106.5.1 Almost Reductive Lie Algebras 3236
106.5.2 Cartan-Type Lie Algebras 3239
106.5.3 Melikian Lie Algebras 3244
106.6 Construction of Elements *3245*
106.6.1 Construction of Elements of Structure Constant Algebras 3246
106.6.2 Construction of Matrix Elements 3246
106.7 Construction of Subalgebras, Ideals and Quotients *3247*
106.8 Operations on Lie Algebras *3249*
106.8.1 Basic Invariants 3252
106.8.2 Changing Base Rings 3253
106.8.3 Bases 3253
106.8.4 Operations for Semisimple and Reductive Lie Algebras 3254
106.9 Operations on Subalgebras and Ideals *3261*
106.9.1 Standard Ideals and Subalgebras 3262
106.9.2 Cartan and Toral Subalgebras 3263
106.9.3 Standard Series 3265
106.9.4 The Lie Algebra of Derivations 3267
106.10 Properties of Lie Algebras and Ideals *3268*
106.11 Operations on Elements *3270*
106.11.1 Indexing 3271
106.12 The Natural Module *3272*
106.13 Operations for Matrix Lie Algebras *3273*

106.14 Homomorphisms 3273
106.15 Automorphisms of Classical-type Reductive Algebras 3274
106.16 Restrictable Lie Algebras 3275
106.17 Universal Enveloping Algebras 3277
106.17.1 Background 3277
106.17.2 Construction of Universal Enveloping Algebras 3278
106.17.3 Related Structures 3279
106.17.4 Elements of Universal Enveloping Algebras 3279
106.18 Solvable and Nilpotent Lie Algebras Classification 3282
106.18.1 The List of Solvable Lie Algebras 3282
106.18.2 Comments on the Classification over Finite Fields 3283
106.18.3 The List of Nilpotent Lie Algebras 3284
106.18.4 Intrinsics for Working with the Classifications 3285
106.19 Semisimple Subalgebras of Simple Lie Algebras 3289
106.20 Nilpotent Orbits in Simple Lie Algebras 3291
106.21 Bibliography 3295

107 KAC-MOODY LIE ALGEBRAS 3297

107.1 Introduction 3299
107.2 Generalized Cartan Matrices 3300
107.3 Affine Kac-Moody Lie Algebras 3301
107.3.1 Constructing Affine Kac-Moody Lie Algebras 3301
107.3.2 Properties of Affine Kac-Moody Lie Algebras 3302
107.3.3 Constructing Elements of Affine Kac-Moody Lie Algebras 3303
107.3.4 Properties of Elements of Affine Kac-Moody Lie Algebras 3304
107.4 Bibliography 3305

108 QUANTUM GROUPS 3307

108.1 Introduction 3309
108.2 Background 3309
108.2.1 Gaussian Binomials 3309
108.2.2 Quantized Enveloping Algebras 3310
108.2.3 Representations of $U_q(L)$ 3311
108.2.4 PBW-type Bases 3311
108.2.5 The $\mathbf{Z}$-form of $U_q(L)$ 3312
108.2.6 The Canonical Basis 3313
108.2.7 The Path Model 3314
108.3 Gauss Numbers 3315
108.4 Construction 3316
108.5 Related Structures 3317
108.6 Operations on Elements 3318
108.7 Representations 3320
108.8 Hopf Algebra Structure 3323
108.9 Automorphisms 3324
108.10 Kashiwara Operators 3326
108.11 The Path Model 3326
108.12 Elements of the Canonical Basis 3329
108.13 Homomorphisms to the Universal Enveloping Algebra 3331
108.14 Bibliography 3332

109 GROUPS OF LIE TYPE 3333

109.1 Introduction *3337*
109.1.1 The Steinberg Presentation 3337
109.1.2 Bruhat Normalisation 3337
109.1.3 Twisted Groups of Lie type 3338
109.2 Constructing Groups of Lie Type *3338*
109.2.1 Split Groups 3338
109.2.2 Galois Cohomology 3341
109.2.3 Twisted Groups 3345
109.3 Operations on Groups of Lie Type *3347*
109.4 Properties of Groups of Lie Type *3351*
109.5 Constructing Elements *3352*
109.6 Operations on Elements *3354*
109.6.1 Basic Operations 3354
109.6.2 Decompositions 3355
109.6.3 Conjugacy and Cohomology 3356
109.7 Properties of Elements *3357*
109.8 Roots, Coroots and Weights *3357*
109.8.1 Accessing Roots and Coroots 3357
109.8.2 Reflections 3360
109.8.3 Operations and Properties for Root and Coroot Indices 3360
109.8.4 Weights 3361
109.9 Building Groups of Lie Type *3361*
109.10 Automorphisms *3363*
109.10.1 Basic Functionality 3363
109.10.2 Constructing Special Automorphisms 3364
109.10.3 Operations and Properties of Automorphisms 3365
109.11 Algebraic Homomorphisms *3366*
109.12 Twisted Tori *3366*
109.13 Sylow Subgroups *3368*
109.14 Representations *3369*
109.15 Bibliography *3371*

110 REPRESENTATIONS OF LIE GROUPS AND ALGEBRAS . . 3373

110.1 Introduction *3375*
110.1.1 Highest Weight Modules 3375
110.1.2 Toral Elements 3376
110.1.3 Other Highest Weight Representations 3376
110.2 Constructing Weight Multisets *3377*
110.3 Constructing Representations *3378*
110.3.1 Lie Algebras 3378
110.3.2 Groups of Lie Type 3382
110.4 Operations on Weight Multisets *3384*
110.4.1 Basic Operations 3384
110.4.2 Conversion Functions 3387
110.4.3 Calculating with Representations 3388
110.5 Operations on Representations *3398*
110.5.1 Lie Algebras 3398
110.5.2 Groups of Lie Type 3402
110.6 Other Functions for Representation Decompositions *3403*
110.6.1 Operations Related to the Symmetric Group 3407
110.6.2 FusionRules 3408
110.7 Subgroups of Small Rank *3409*
110.8 Subalgebras of su(d) *3410*
110.9 Bibliography *3412*

PART XIII
LIE THEORY

100	INTRODUCTION TO LIE THEORY	3033
101	COXETER SYSTEMS	3039
102	ROOT SYSTEMS	3063
103	ROOT DATA	3085
104	COXETER GROUPS	3137
105	REFLECTION GROUPS	3177
106	LIE ALGEBRAS	3209
107	KAC-MOODY LIE ALGEBRAS	3297
108	QUANTUM GROUPS	3307
109	GROUPS OF LIE TYPE	3333
110	REPRESENTATIONS OF LIE GROUPS AND ALGEBRAS	3373

100 INTRODUCTION TO LIE THEORY

100.1 Descriptions of Coxeter Groups 3035

100.2 Root Systems and Root Data 3036

100.3 Coxeter and Reflection Groups 3036

100.4 Lie Algebras and Groups of Lie Type 3037

100.5 Highest Weight Representations 3037

100.6 Universal Enveloping Algebras and Quantum Groups . . . 3037

100.7 Bibliography 3038

Chapter 100
INTRODUCTION TO LIE THEORY

A number of structures from Lie theory and the theory of Coxeter groups can be handled by MAGMA. Specifically, facilities are provided for:

1. Coxeter matrices, Coxeter graphs, Cartan matrices, Dynkin diagrams, and Cartan's naming system for Coxeter groups;
2. Finite root systems and finite root data;
3. Coxeter groups in three different formats: as finitely presented groups, as permutation groups, and as reflection groups;
4. Complex reflection groups;
4. Lie algebras, given as structure constant algebras, matrix algebras, or finitely generated algebras;
5. Groups of Lie type (connected reductive algebraic groups);
5. Representations of Lie algebras and groups of Lie type;
6. Universal enveloping algebras and Quantum groups.

100.1 Descriptions of Coxeter Groups

A *Coxeter system* is a group G with finite generating set $S = \{s_1, \ldots, s_n\}$, defined by the power relations $s_i^2 = 1$ for $i = 1, \ldots, n$ and braid relations

$$s_i s_j s_i \cdots = s_j s_i s_j \cdots$$

for $i, j = 1, \ldots, n$ with $i < j$, where each side of this relation has length $m_{ij} \geq 2$. Although traditionally $m_{ij} = \infty$ signifies that the corresponding relation is omitted, for technical reasons, we use $m_{ij} = 0$ instead. Set $m_{ji} = m_{ij}$ and $m_{ii} = 1$. The group G is called a *Coxeter group* and S is called the set of *Coxeter generators*. Since every group in MAGMA has a preferred generating set, no distinction is made between a Coxeter system and its Coxeter group.

Due to the importance and ubiquity of Coxeter groups, a number of different ways of describing these groups and their reflections have been developed. Functions for manipulating these descriptions are described in Chapter 101.

Coxeter groups are usually described by a *Coxeter matrix* $M = (m_{ij})_{i,j=1}^n$, or by a *Coxeter graph* with vertices $1, \ldots, n$ and an edge connecting i and j labeled by m_{ij} whenever $m_{ij} \geq 3$.

Coxeter systems are mainly important because they provide presentations for the real reflection groups. A *Cartan matrix* describes a particular reflection representation of a

Coxeter group. In certain cases, such a representation can be described by an integer-labelled digraph, called the *Dynkin digraph* (this is equivalent to a *Dynkin diagram*, but we have modified the definition for technical reasons).

For finite and affine Coxeter groups, the naming system due to Cartan is also used. Hyperbolic Coxeter groups of degree larger than 3 are numbered.

100.2 Root Systems and Root Data

A (real) reflection is an automorphism of a real vector space that acts as negation on a one-dimensional subspace while fixing a hyperplane pointwise. The subspace is described by a vector called the *root*, while the hyperplane is described as the kernel of an element of the dual space called the *coroot*.

A root system is a collection of root/coroot pairs that is closed under the action of the corresponding reflections. Only finite root systems are supported at the present time. A root system gives a much more detailed description of a reflection representation of a finite Coxeter group.

Root systems are used to classify the *semisimple Lie algebras*. The closely related concept of a root datum is used to classify the *groups of Lie type*.

This is described in Chapters 102 and 103.

100.3 Coxeter and Reflection Groups

Three different methods are provided for computing with a Coxeter group: the Coxeter presentation, the permutation representation on roots, or a reflection representation.

For most purposes, the presentation will be the most useful of these descriptions. The standard normal form is used for elements (the lexicographically least word of minimal length). Robert Howlett has implemented his highly efficient method for normalising and multiplying elements, based on ideas from [BH93].

If the Coxeter group is finite, it is often better to use the permutation representation. Note that elements are represented as permutations on the set of roots. This is not the minimal degree representation, but is more useful in many cases.

Finally, Coxeter groups can be represented as a reflection group over the reals (in practice over a number field, since the reals are not infinite precision). Although functions are provided for creating reflection groups over an arbitrary field, fewer facilities are available for such groups. In addition, functions are provided to construct all the finite *complex* reflection groups.

Efficient functions are provided for converting between these three forms of Coxeter group.

This is described in Chapters 104 and 105.

100.4 Lie Algebras and Groups of Lie Type

Lie algebras can be constructed in three different ways in MAGMA: as structure constant algebras, as Lie matrix algebras, or as finitely presented algebras. Most of our functionality is for algebras of finite dimension over a field. Algorithms designed and implemented by de Graaf [dG00] are available for determining the structure of a Lie algebra. In particular, if the algebra is reductive, its root system and its highest-weight representations can be determined.

We provide functionality for computing with groups of Lie type (i.e. reductive algebraic groups and their split (untwisted) groups, given by the Steinberg presentation. A canonical form for words in this group, and algorithms for computing with these words are given in [CMT04, CHM08]. Twisted groups are given by a modified version of this presentation using Galois cohomology [Hal05]. Efficient algorithms have been implemented for arithmetic with the Steinberg presentation and for converting between this presentation and matrix representations over the base field. Note that these presentations are not in the category `GrpFP` since the generators are parametrised by field elements and so the groups involved are not necessarily finitely generated.

This is described in Chapter 106.

100.5 Highest Weight Representations

The representations of Lie algebras and connected reductive Lie groups are classified by *highest weights*. In addition to being able to construct these representations [dG01], we can compute the combinatorics of their weights. This includes all the functionality of the LiE system [vLCL92]. This is described in Chapter 110.

100.6 Universal Enveloping Algebras and Quantum Groups

Given a semisimple Lie algebra over a field of characteristic zero, we can construct an integral basis for its universal enveloping algebra. Functionality for computing in these algebras is described in Section 106.17. Moreover, we provide functionality for computing in their quantised versions, which are called quantum groups. This is described in Chapter 108.

100.7 Bibliography

[**BH93**] Brigitte Brink and Robert B. Howlett. A finiteness property and an automatic structure for Coxeter groups. *Math. Ann.*, 296(1):179–190, 1993.

[**CHM08**] Arjeh M. Cohen, Sergei Haller, and Scott H. Murray. Computing in unipotent and reductive algebraic groups. *LMS J. Comput. Math.*, 11:343–366, 2008.

[**CMT04**] Arjeh M. Cohen, Scott H. Murray, and D. E. Taylor. Computing in groups of Lie type. *Math. Comp.*, 73(247):1477–1498, 2004.

[**dG00**] W.A. de Graaf. *Lie Algebras: Theory and Algorithms*. Number 56 in North-Holland Mathematical Library. Elsevier, 2000.

[**dG01**] W. A. de Graaf. Constructing representations of split semisimple Lie algebras. *J. Pure Appl. Algebra*, 164(1-2):87–107, 2001. Effective methods in algebraic geometry (Bath, 2000).

[**Hal05**] Sergei Haller. *Computing Galois Cohomology and Forms of Linear Algebraic Groups*. Phd thesis, Technical University of Eindhoven, 2005.

[**vLCL92**] M.A.A. van Leeuwen, A.M. Cohen, and B. Lisser. *LiE, A package for Lie Group Computations*. CAN, Amsterdam, 1992.

101 COXETER SYSTEMS

101.1 Introduction 3041

101.2 Coxeter Matrices 3041

IsCoxeterMatrix(M) 3041
CoxeterMatrix(G) 3042
CoxeterMatrix(C) 3042
CoxeterMatrix(D) 3042
IsCoxeterIsomorphic(M1, M2) 3042
CoxeterGroupOrder(M) 3042
CoxeterGroupFactoredOrder(M) 3042
IsCoxeterIrreducible(M) 3042
IsSimplyLaced(M) 3042

101.3 Coxeter Graphs 3043

IsCoxeterGraph(G) 3043
CoxeterGraph(M) 3043
CoxeterGraph(C) 3043
CoxeterGraph(D) 3043
CoxeterGroupOrder(G) 3043
CoxeterGroupFactoredOrder(G) 3043
IsSimplyLaced(G) 3044

101.4 Cartan Matrices 3045

IsCartanMatrix(C) 3045
CartanMatrix(M) 3045
CartanMatrix(G) 3045
CartanMatrix(D) 3046
IsCoxeterIsomorphic(C1, C2) 3046
IsCartanEquivalent(C1, C2) 3047
NumberOfPositiveRoots(C) 3047
NumPosRoots(C) 3047
CoxeterGroupOrder(C) 3047
CoxeterGroupFactoredOrder(C) 3047
FundamentalGroup(C) 3047
IsCoxeterIrreducible(C) 3048
IsCrystallographic(C) 3048
IsSimplyLaced(C) 3048

101.5 Dynkin Digraphs 3048

IsDynkinDigraph(D) 3049
DynkinDigraph(C) 3049
CoxeterGroupOrder(D) 3049
CoxeterGroupFactoredOrder(D) 3049
FundamentalGroup(D) 3049
IsSimplyLaced(D) 3049

101.6 Finite and Affine Coxeter Groups 3050

IsCoxeterFinite(M) 3052
IsCoxeterFinite(G) 3052
IsCoxeterFinite(C) 3052
IsCoxeterFinite(D) 3052
IsCoxeterFinite(N) 3052
IsCoxeterAffine(M) 3052
IsCoxeterAffine(G) 3052
IsCoxeterAffine(C) 3052
IsCoxeterAffine(D) 3052
IsCoxeterAffine(N) 3052
CoxeterMatrix(N) 3052
CoxeterGraph(N) 3052
CartanMatrix(N) 3053
DynkinDigraph(N) 3053
IrreducibleCoxeterMatrix(X, n) 3054
IrreducibleCoxeterGraph(X, n) 3054
IrreducibleCartanMatrix(X, n) 3054
IrreducibleDynkinDigraph(X, n) 3054
IsCoxeterIsomorphic(N1, N2) 3055
IsCartanEquivalent(N1, N2) 3055
IsSimplyLaced(N) 3055
CoxeterGroupOrder(N) 3055
CoxeterGroupFactoredOrder(N) 3055
NumberOfPositiveRoots(N) 3056
NumPosRoots(N) 3056
FundamentalGroup(N) 3056
CartanName(M) 3056
CartanName(G) 3056
CartanName(C) 3056
CartanName(D) 3056
DynkinDiagram(M) 3057
DynkinDiagram(G) 3057
DynkinDiagram(C) 3057
DynkinDiagram(D) 3057
DynkinDiagram(N) 3057
CoxeterDiagram(M) 3057
CoxeterDiagram(G) 3057
CoxeterDiagram(C) 3057
CoxeterDiagram(D) 3057
CoxeterDiagram(N) 3057

101.7 Hyperbolic Groups 3058

IsCoxeterHyperbolic(M) 3058
IsCoxeterCompactHyperbolic(M) 3058
IsCoxeterHyperbolic(G) 3058
IsCoxeterCompactHyperbolic(G) 3058
HyperbolicCoxeterMatrix(i) 3058
HyperbolicCoxeterGraph(i) 3058

101.8 Related Structures 3059

RootSystem(M) 3059
RootSystem(G) 3059
RootSystem(C) 3059
RootSystem(D) 3059
RootSystem(N) 3059
RootDatum(C) 3059
RootDatum(M) 3059
RootDatum(G) 3059
RootDatum(D) 3059
RootDatum(N) 3059
CoxeterGroup(GrpFPCox, M) 3060
CoxeterGroup(GrpFPCox, G) 3060

CoxeterGroup(GrpFPCox, C) 3060
CoxeterGroup(GrpFPCox, D) 3060
CoxeterGroup(GrpFPCox, N) 3060
CoxeterGroup(GrpPermCox, M) 3060
CoxeterGroup(GrpPermCox, G) 3060
CoxeterGroup(GrpPermCox, C) 3060
CoxeterGroup(GrpPermCox, D) 3060
CoxeterGroup(GrpPermCox, N) 3060
CoxeterGroup(M) 3060
CoxeterGroup(G) 3060
CoxeterGroup(C) 3060
CoxeterGroup(D) 3060
CoxeterGroup(N) 3060
ReflectionGroup(M) 3060
ReflectionGroup(G) 3060
ReflectionGroup(C) 3060
ReflectionGroup(D) 3060
ReflectionGroup(N) 3060
LieAlgebra(C, k) 3061
LieAlgebra(D, k) 3061
LieAlgebra(N, k) 3061
MatrixLieAlgebra(C, k) 3061
MatrixLieAlgebra(D, k) 3061
MatrixLieAlgebra(N, k) 3061
GroupOfLieType(C, k) 3061
GroupOfLieType(D, k) 3061
GroupOfLieType(N, k) 3061

101.9 Bibliography 3061

Chapter 101
COXETER SYSTEMS

101.1 Introduction

The functions in this chapter handle basic descriptions of Coxeter systems. A *Coxeter system* is a group G with finite generating set $S = \{s_1, \ldots, s_n\}$, defined by relations $s_i^2 = 1$ for $i = 1, \ldots, n$ and

$$s_i s_j s_i \cdots = s_j s_i s_j \cdots$$

for $i, j = 1, \ldots, n$ with $i < j$, where each side of this relation has length $m_{ij} \geq 2$. Traditionally, $m_{ij} = \infty$ signifies that the corresponding relation is omitted but for technical reasons $m_{ij} = 0$ is used in MAGMA instead. The group G is called a *Coxeter group* and S is called the set of *Coxeter generators*. Since every group in MAGMA has a preferred generating set, no distinction is made between a Coxeter system and its Coxeter group. See [Bou68] for more details on the theory of Coxeter groups.

The *rank* of the Coxeter system is $n = |S|$. A Coxeter system is said to be *reducible* if there is a proper subset I of $\{1, \ldots, n\}$ such that $m_{ij} = 2$ or $m_{ji} = 2$ whenever $i \in I$ and $j \notin I$. In this case, G is an (internal) direct product of the Coxeter subgroups $W_I = \langle s_i \mid i \in I \rangle$ and $W_{I^c} = \langle s_i \mid i \notin I \rangle$. Note that an *irreducible* Coxeter group may still be a nontrivial direct product of abstract subgroups (for example, $W(G_2) \cong S_2 \times S_3$). Two Coxeter systems are *Coxeter isomorphic* (or *graph isomorphic*) if there is a group isomorphism between them which takes Coxeter generators to Coxeter generators. In other words, the two groups are the same modulo renumbering of the generators.

Coxeter groups and their representations as reflection groups have a number of useful descriptions. In this chapter, Coxeter matrices, Coxeter graphs, Cartan matrices, and Dynkin digraphs will be discussed. The classification of finite and affine Coxeter groups provides a naming system for these groups. In Chapters 102 and 103, finite root systems and root data, which provide a more detailed description of finite Coxeter groups, are discussed. Coxeter groups themselves are discussed in Chapter 104; reflection representations of Coxeter groups are discussed in Chapter 105.

101.2 Coxeter Matrices

A Coxeter system is defined by the numbers $m_{ij} \in \{2, 3, \ldots, \infty\}$ for $i, j = 1, \ldots n$ and $i < j$, as in the previous section. Setting $m_{ji} = m_{ij}$ and $m_{ii} = 1$, yields a matrix $M = (m_{ij})_{i,j=1}^n$ that is called the *Coxeter matrix*.

Since ∞ is not an integer in MAGMA, it will be represented by 0 in Coxeter matrices.

`IsCoxeterMatrix(M)`

Returns `true` if, and only if, the matrix M is the Coxeter matrix of some Coxeter group.

`CoxeterMatrix(G)`

`CoxeterMatrix(C)`

`CoxeterMatrix(D)`

The Coxeter matrix corresponding to a Coxeter graph G, Cartan matrix C, or Dynkin digraph D.

Example H101E1

```
> M := SymmetricMatrix([1, 3,1, 2,3,1]);
> M;
[1 3 2]
[3 1 3]
[2 3 1]
> IsCoxeterMatrix(M);
true
```

`IsCoxeterIsomorphic(M1, M2)`

Returns `true` if and only if the Coxeter matrices M_1 and M_2 give rise to isomorphic Coxeter systems. If so, a sequence giving the permutation of the underlying basis which takes M_1 to M_2 is also returned.

`CoxeterGroupOrder(M)`

`CoxeterGroupFactoredOrder(M)`

The (factored) order of the Coxeter group with Coxeter matrix M.

Example H101E2

```
> M1 := SymmetricMatrix([1, 3,1, 2,3,1]);
> M2 := SymmetricMatrix([1, 3,1, 3,2,1]);
> IsCoxeterIsomorphic(M1, M2);
true [ 2, 1, 3 ]
>
> CoxeterGroupOrder(M1);
24
```

`IsCoxeterIrreducible(M)`

Returns `true` if, and only if, the matrix M is the Coxeter matrix of an irreducible Coxeter system. If the Coxeter matrix is reducible, this function also returns a nontrivial subset I of $\{1, \ldots, n\}$ such that $m_{ij} = 2$ whenever $i \in I$, $j \notin I$.

`IsSimplyLaced(M)`

Returns `true` if, and only if, the Coxeter matrix M is simply laced, i.e. all its entries are 1, 2, or 3.

Example H101E3

```
> M := SymmetricMatrix([1, 3,1, 2,3,1]);
> IsCoxeterIrreducible(M);
true
> M := SymmetricMatrix([1, 2,1, 2,3,1]);
> IsCoxeterIrreducible(M);
false { 1 }
```

101.3 Coxeter Graphs

A *Coxeter graph* is an undirected labelled graph describing a Coxeter system. Suppose a Coxeter system has Coxeter matrix $M = (m_{ij})_{i,j=1}^{n}$. Then the Coxeter graph has vertices $1, \ldots, n$; whenever $m_{ij} > 2$ there is an edge connecting i and j labeled by the value of m_{ij}. When $m_{ij} = 3$, the label is usually omitted.

Since ∞ is not an integer, it will be represented by 0 in our Coxeter graphs. Clearly a Coxeter graph must be standard, i.e. its vertices must be the integers $1, 2, \ldots, n$ for some n. A Coxeter system is irreducible if, and only if, its Coxeter graph is connected. Two Coxeter graphs give rise to Coxeter isomorphic groups if, and only if, they are isomorphic as labelled graphs. See Chapter 155 for more information on graphs.

`IsCoxeterGraph(G)`

Returns `true` if, and only if, the graph G is the Coxeter graph of some Coxeter group.

`CoxeterGraph(M)`

`CoxeterGraph(C)`

`CoxeterGraph(D)`

The Coxeter graph corresponding to a Coxeter matrix M, Cartan matrix C, or Dynkin digraph D.

`CoxeterGroupOrder(G)`

`CoxeterGroupFactoredOrder(G)`

The (factored) order of the Coxeter group with Coxeter graph G.

Example H101E4

```
> G := PathGraph(4);
> AssignLabel(G, 1,2, 4);
> AssignLabel(G, 3,4, 4);
> IsCoxeterGraph(G);
true
> CoxeterGroupOrder(G);
Infinity
>
> M := SymmetricMatrix([1, 3,1, 2,5,1]);
> G := CoxeterGraph(M);
> Labels(EdgeSet(G));
[ undef, 5 ]
```

`IsSimplyLaced(G)`

Returns `true` if, and only if, the Coxeter graph G is simply laced, i.e. unlabelled.

Example H101E5

```
> G := PathGraph(2);
> IsSimplyLaced(G);
true
> AssignLabel(G, 1,2, 6);
> IsSimplyLaced(G);
false
```

101.4 Cartan Matrices

A *Cartan matrix* is a real valued matrix $C = (c_{ij})_{i,j=1}^{n}$ satisfying the properties:

1. $c_{ii} = 2$;
2. $c_{ij} \leq 0$ for $i \neq j$;
3. $c_{ij} = 0$ if, and only if, $c_{ji} = 0$; and
4. if $n_{ij} := c_{ij}c_{ji} < 4$, then $n_{ij} = 4\cos^2(\pi/m_{ij})$ for some integer $m_{ij} \geq 2$.

In MAGMA, Cartan matrices can be defined over the integer ring (Chapter 18), the rational field (Chapter 20), number fields (Chapter 35), and cyclotomic fields (Chapter 37). The real field (Chapter 25) is *not* allowed since it is not infinite precision. A Cartan matrix is called *crystallographic* if all its entries are integers.

Given a Cartan matrix, the corresponding Coxeter matrix $M = (m_{ij})_{i,j=1}^{n}$ is defined by $m_{ii} = 1$; m_{ij} as in (4) if $n_{ij} < 4$; $m_{ij} = \infty$ (ie, 0) if $n_{ij} \geq 4$. The significance of Cartan matrices is due to the following construction: Let X be a real inner-product space with basis $\alpha_1, \ldots, \alpha_n$. Take the unique basis $\alpha_1^\star, \ldots, \alpha_n^\star$ for X such that $(\alpha_i, \alpha_j^\star) = c_{ij}$. Let s_i be the reflection in α_i and $\alpha_i^\star$, i.e. $s_i : V \to V$ is defined by $vs_i = v - (v, \alpha_i^\star)\alpha_i$. Then the group generated by $s_1, \ldots, s_n$ is a Coxeter group with Coxeter matrix M. In other words, a Cartan matrix specifies a faithful representation of the Coxeter group as a real reflection group. For more details on reflection groups see Chapter 105.

`IsCartanMatrix(C)`

`RealInjection`	ANY	*Default :* `false`

Returns `true` if, and only if, the matrix C is a Cartan matrix.

Number field elements and cyclotomic field elements do not have a natural identification with real numbers. The `RealInjection` flag allows the user to provide one. If the base field of C is a number field, the flag should be an injection into the real field; if the base field is cyclotomic, the flag should be an injection into the complex field taking real values on the entries of C. If no real injection is given, conditions (2) and (4) of the definition are not checked.

`CartanMatrix(M)`

`CartanMatrix(G)`

`Symmetric`	BOOLELT	*Default :* `false`
`BaseField`	MONSTGELT	*Default :* "*NumberField*"

A Cartan matrix corresponding to the Coxeter matrix M or Coxeter graph G. Note that the Cartan matrix of a Coxeter system is not unique. By default this function returns the Cartan matrix with $c_{ij} = -4\cos^2(\pi/m_{ij})$, $c_{ji} = -1$ when $m_{ij} \neq 2$ and $i < j$. This matrix is crystallographic whenever there exists a crystallographic Cartan matrix corresponding to M.

If the `Symmetric` flag is set `true`, the symmetric Cartan matrix with

$$c_{ij} = c_{ji} = -2\cos(\pi/m_{ij})$$

is returned.

The `BaseField` flag determines the field over which the Cartan matrix is defined. If the matrix is crystallographic however, it is defined over the integers regardless of the value of this flag. The possible values are:

1. `"NumberField"`: An algebraic number field. This is the default. See Chapter 35.
2. `"Cyclotomic"` or `"SparseCyclotomic"`: A cyclotomic field with the sparse representation for elements. See Chapter 37.
3. `"DenseCyclotomic"`: A cyclotomic field with the dense representation for elements. See Chapter 37.

`CartanMatrix(D)`

The crystallographic Cartan matrix corresponding to the Dynkin digraph D.

Example H101E6

```
> C := Matrix(2,2, [ 2,-3, -1,2 ]);
> C;
> IsCartanMatrix(C);
true
> CoxeterMatrix(C);
[1 6]
[6 1]
>
> G := PathGraph(4);
> AssignLabel(G, 1,2, 4);
> AssignLabel(G, 3,4, 4);
> CartanMatrix(G);
[ 2 -2  0  0]
[-1  2 -1  0]
[ 0 -1  2 -2]
[ 0  0 -1  2]
> CartanMatrix(G : Symmetric, BaseField := "Cyclotomic");
[2 zeta(8)_8^3 - zeta(8)_8 0 0]
[zeta(8)_8^3 - zeta(8)_8 2 -1 0]
[0 -1 2 zeta(8)_8^3 - zeta(8)_8]
[0 0 zeta(8)_8^3 - zeta(8)_8 2]
```

`IsCoxeterIsomorphic(C1, C2)`

Tests if the Cartan matrices C_1 and C_2 give rise to isomorphic Coxeter systems; i.e., their Coxeter matrices are equal modulo a permutation of the underlying basis. If `true`, a sequence giving the permutation of the underlying basis which takes the Coxeter matrix of C_1 to the Coxeter matrix of C_2 is also returned.

`IsCartanEquivalent(C1, C2)`

Returns `true` if, and only if, the crystallographic Cartan matrices C_1 and C_2 are Cartan equivalent, i.e. they are equal modulo permutation of the underlying basis. If so, a sequence giving the permutation of the underlying basis which takes C_1 to C_2 is also returned.

Example H101E7

Cartan equivalence is a stronger condition than Coxeter isomorphism.

```
> C1 := Matrix(2,2, [ 2,-2, -2,2 ]);
> C2 := Matrix(2,2, [ 2,-1, -5,2 ]);
> IsCoxeterIsomorphic(C1, C2);
true [ 1, 2 ]
> IsCartanEquivalent(C1, C2);
false
```

`NumberOfPositiveRoots(C)`

`NumPosRoots(C)`

The number of positive roots of the root system with Cartan matrix C. See Subsection 102.1.3 for the definition of positive roots.

`CoxeterGroupOrder(C)`

`CoxeterGroupFactoredOrder(C)`

The (factored) order of the Coxeter group with Cartan matrix C.

`FundamentalGroup(C)`

The fundamental group of the crystallographic Cartan matrix C, i.e. $\mathbf{Z}^n/\Gamma$ where n is the degree of C and Γ is the lattice generated by the rows of C. The natural mapping $\mathbf{Z}^n \to \mathbf{Z}^n/\Gamma$ is the second returned value.

Example H101E8

```
> C := CartanMatrix(PathGraph(4));
> FundamentalGroup(C);
Abelian Group isomorphic to Z/5
Defined on 1 generator
Relations:
    5*$.1 = 0
Mapping from: Standard Lattice of rank 4 and degree 4 to Abelian Group
isomorphic to Z/5
Defined on 1 generator
Relations:
    5*$.1 = 0
```

`IsCoxeterIrreducible(C)`

Returns `true` if, and only if, C is the Cartan matrix of an irreducible Coxeter system. If the Coxeter matrix is reducible, this function also returns a nontrivial subset I of $\{1, \ldots, n\}$ such that $m_{ij} = 2$ (i.e. $c_{ij} = 0$) whenever $i \in I$, $j \notin I$.

`IsCrystallographic(C)`

Returns `true` if, and only if, the Cartan matrix C is crystallographic, i.e. C has integral entries.

`IsSimplyLaced(C)`

Returns `true` if, and only if, the Cartan matrix C is simply laced, i.e. all the entries in its Coxeter matrix are 1, 2, or 3.

Example H101E9

```
> C := Matrix(2,2, [ 2,-2, -2,2 ]);
> IsCoxeterIrreducible(C);
true
> IsCrystallographic(C);
true
> IsSimplyLaced(C);
false
```

101.5 Dynkin Digraphs

A *Dynkin digraph* is a directed labelled graph describing a *crystallographic* Cartan matrix $C = (c_{ij})_{i,j=1}^{n}$. The Dynkin digraph has vertices $1, \ldots, n$; whenever $c_{ij} < 0$ there is an edge from i to j labeled by the value $-c_{ij}$. When $c_{ij} = -1$, the label is usually omitted.

In the literature, the term *Dynkin diagram* is used, but here this will be reserved for a printed display of the Dynkin digraph (or Coxeter graph) corresponding to a finite or affine Coxeter group (see Section 101.6 below). For convenience, Dynkin digraphs have labelled edges rather than multiple edges.

Clearly a Dynkin digraph must be standard, i.e. its vertices must be the integers $1, 2, \ldots, n$ for some n. A Dynkin digraph has an edge from i to j if, and only if, it has an edge from j to i (although the labels may be different). Hence strong and weak connectivity are equivalent for these graphs. The Coxeter system is irreducible if, and only if, the Dynkin digraph is connected. Two Dynkin digraphs give rise to Cartan equivalent Cartan matrices if they are isomorphic as labelled digraphs. See Chapter 155 for more information on digraphs.

Note that functions are not given for computing the Dynkin digraph of a Coxeter matrix or Coxeter graph, since a particular choice of crystallographic Cartan matrix is required.

`IsDynkinDigraph(D)`

Returns `true` if, and only if, the digraph D is the Dynkin digraph of some crystallographic Cartan matrix.

`DynkinDigraph(C)`

The Dynkin digraph of the crystallographic Cartan matrix C.

`CoxeterGroupOrder(D)`

`CoxeterGroupFactoredOrder(D)`

The (factored) order of the Coxeter group with Dynkin digraph D.

`FundamentalGroup(D)`

The fundamental group of the Dynkin digraph D, i.e. $\mathbf{Z}^n/\Gamma$ where n is the degree of D and Γ is the lattice generated by the rows of the corresponding Cartan matrix. The natural mapping $\mathbf{Z}^n \to \mathbf{Z}^n/\Gamma$ is the second returned value.

`IsSimplyLaced(D)`

Returns `true` if, and only if, the Dynkin digraph D is simply laced, i.e. unlabelled.

Example H101E10

```
> D := Digraph< 4 | <1,{2,3,4}>, <2,{1}>, <3,{1}>, <4,{1}> >;
> AssignLabel(D, 1,2, 2);
> AssignLabel(D, 1,3, 5);
> IsDynkinDigraph(D);
true
> CartanMatrix(D);
[ 2 -2 -5 -1]
[-1  2  0  0]
[-1  0  2  0]
[-1  0  0  2]
> FundamentalGroup(D);
Abelian Group isomorphic to Z/2 + Z/8
Defined on 2 generators
Relations:
    2*$.1 = 0
    8*$.2 = 0
Mapping from: Standard Lattice of rank 4 and degree 4 to Abelian Group
isomorphic to Z/2 + Z/8
Defined on 2 generators
Relations:
    2*$.1 = 0
    8*$.2 = 0
```

101.6 Finite and Affine Coxeter Groups

Functions related to the classification of finite and affine Coxeter groups are described in this section. This classification is due to Cartan [Car52] and Coxeter [Cox34].

An affine reflection group is a group generated by reflections in affine space (in other words, real reflections in a hyperplane that does not necessarily pass through the origin). A Coxeter group is called *affine* if it is infinite and it has a representation as a discrete, properly acting, affine reflection group (see [Bou68] for more details on discreteness and proper action). Note that a Coxeter group is finite if, and only if, it has a representation as a discrete, properly acting group of reflections of the sphere; hence finite Coxeter groups are sometimes called *spherical*.

A Coxeter group is finite if, and only if, all its irreducible components are finite; a Coxeter group is affine if, and only if, all its irreducible components are finite or affine, and at least one component is affine. So it suffices to classify irreducible Coxeter groups.

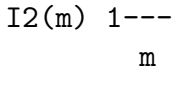
The Dynkin diagrams of the irreducible finite crystallographic Coxeter groups are:

```
An  1---2---3- ... -n        Bn  1---2- ... -(n-1)=>=n

                                                 (n-1)
                                                /
Cn  1---2- ... -(n-1)=<=n    Dn  1---1- ... -(n-2)
                                                \
                                                 n

E6  1---3---4---5---6        E7  1---3---4---5---6---7
            |                            |
            2                            2

E8  1---3---4---5---6---7---8
            |
            2

F4  1---2=>=3---4            G2  1=<=2
                                     3
```

Due to the difficulty of drawing a triple bond with text characters, the edge for G_2 is labelled.

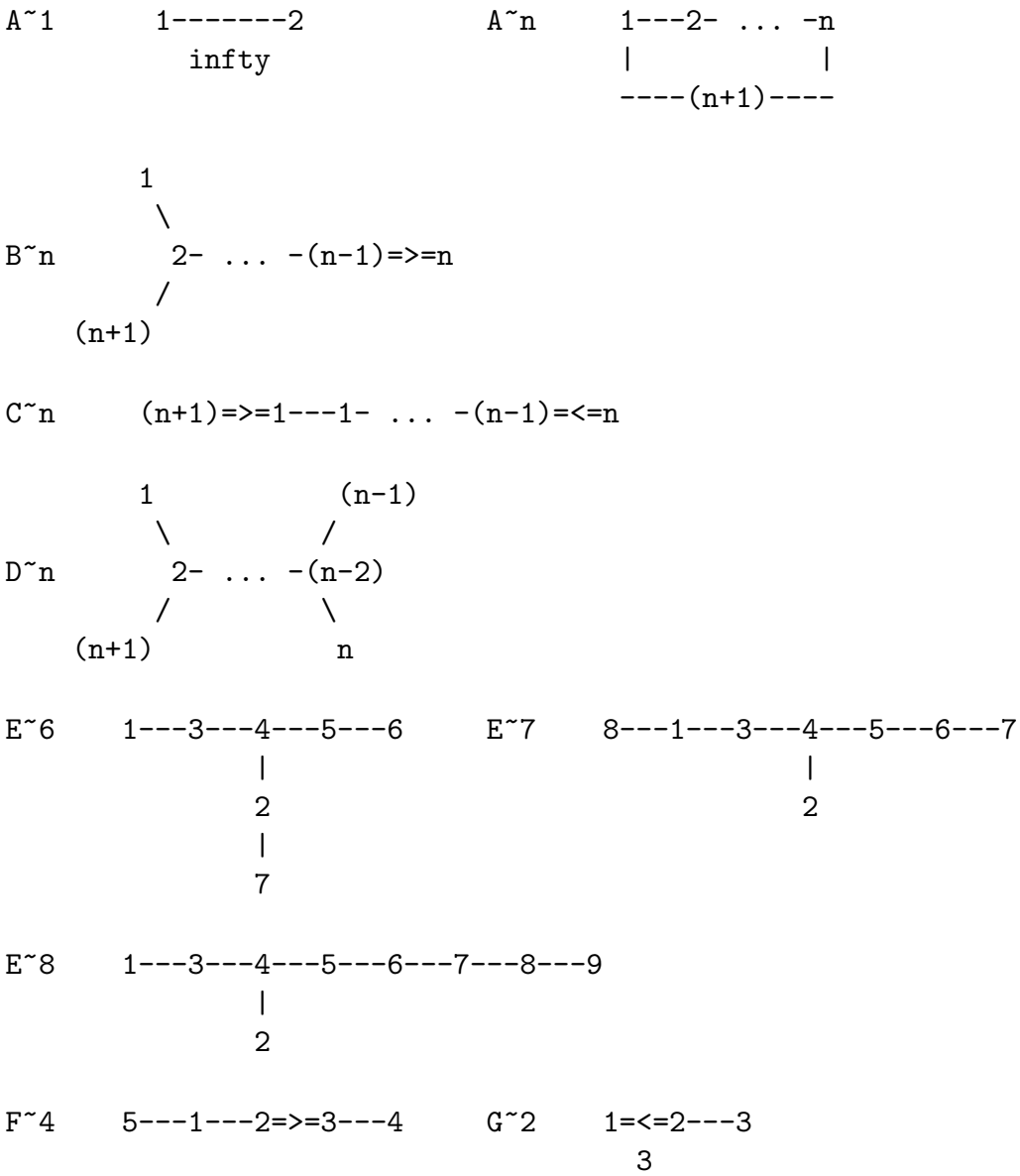
The only irreducible noncrystallographic finite Coxeter groups are H_3, H_4 and $I_2(m)$ for $m = 5$ and $m > 6$. The Coxeter graphs of these groups are:

```
H3  1---2---3                H4  1---2---3---4
      5                            5
```



Note that there is some redundancy in this classification; specifically $A_1 = B_1 = C_1 = D_1$, $A_2 = I_2(3)$, $B_2 = C_2 = I_2(4)$, $D_2 = A_1 + A_1$, $D_3 = A_3$, $G_2 = I_2(6)$. Furthermore, for $n \geq 3$, types B_n and C_n have identical Coxeter matrices but inequivalent crystallographic Cartan matrices for $n > 2$.

All irreducible affine groups are crystallographic. There is one corresponding to each irreducible crystallographic finite group. Their Dynkin diagrams are:



The labels on the vertices of these diagrams show the standard vertex order used in MAGMA, which is consistent with the order used in [Bou68].

`IsCoxeterFinite(M)`
`IsCoxeterFinite(G)`
`IsCoxeterFinite(C)`
`IsCoxeterFinite(D)`
`IsCoxeterFinite(N)`

Returns `true` if, and only if, the corresponding Coxeter group is finite. The input variable can be a Coxeter matrix M, Coxeter graph G, Cartan matrix C, Dynkin digraph D, or Cartan name given by the string N.

`IsCoxeterAffine(M)`
`IsCoxeterAffine(G)`
`IsCoxeterAffine(C)`
`IsCoxeterAffine(D)`
`IsCoxeterAffine(N)`

Returns `true` if, and only if, the corresponding Coxeter group is affine. The input variable can be a Coxeter matrix M, Coxeter graph G, Cartan matrix C, Dynkin digraph D, or Cartan name given by the string N.

Example H101E11

```
> IsCoxeterAffine("A~2");
true
> IsCoxeterAffine("A~2B2");
true
> IsCoxeterAffine("A2B2");
false
> IsCoxeterFinite("A2B2");
true
```

`CoxeterMatrix(N)`

The Coxeter matrix with Cartan name given by the string N.

`CoxeterGraph(N)`

The Coxeter graph with Cartan name given by the string N.

`CartanMatrix(N)`

`Symmetric`	BoolElt	*Default :* `false`
`BaseField`	MonStgElt	*Default :* "*NumberField*"

The Cartan matrix with Cartan name given by the string N. By default, the crystallographic matrix is returned for crystallographic types; otherwise the Cartan matrix with $c_{ij} = -4\cos^2(\pi/m_{ij})$, $c_{ji} = -1$ when $m_{ij} \neq 2$ and $i < j$ is returned.

If the `Symmetric` flag is set `true`, the symmetric Cartan matrix with $c_{ij} = c_{ji} = -2\cos(\pi/m_{ij})$ is returned.

The `BaseField` flag determines the field over which the Cartan matrix is defined. If the matrix is crystallographic however, it is defined over the integers regardless of the value of this flag. The possible values are:

1. `"NumberField"`: An algebraic number field. This is the default. See Chapter 35.
2. `"Cyclotomic"` or `"SparseCyclotomic"`: A cyclotomic field with the sparse representation for elements. See Chapter 37.
3. `"DenseCyclotomic"`: A cyclotomic field with the dense representation for elements. See Chapter 37.

`DynkinDigraph(N)`

The Dynkin digraph with Cartan name given by the string N. The Cartan name must be crystallographic, i.e. it cannot involve types H_3, H_4 and $I_2(m)$.

Example H101E12

```
> CoxeterMatrix("I2(7)");
[1 7]
[7 1]
> CoxeterGraph("A3");
Graph
Vertex  Neighbours
1       2 ;
2       1 3 ;
3       2 ;
> CartanMatrix("H3" : Symmetric);
[   2 -$.1    0]
[-$.1    2   -1]
[   0   -1    2]
> DynkinDigraph("A~2");
Digraph
Vertex  Neighbours
1       2 3 ;
2       1 3 ;
```

```
3       1 2 ;
```

The code for interpreting a string as a Cartan name is quite flexible: letters and numbers must alternate, except in type I where brackets must be used.

```
> M := CoxeterMatrix("A_5B3 c2I2 (5)");
> CartanName(M);
A5 B3 B2 I2(5)
```

`IrreducibleCoxeterMatrix(X, n)`

The irreducible Coxeter matrix with Cartan name X_n (or $I_2(n)$ if $X =$`"I"`).

`IrreducibleCoxeterGraph(X, n)`

The irreducible Coxeter graph with Cartan name X_n (or $I_2(n)$ if $X =$`"I"`).

`IrreducibleCartanMatrix(X, n)`

`Symmetric`	BoolElt	*Default :* `false`
`BaseField`	MonStgElt	*Default :* "*NumberField*"

The irreducible Cartan matrix with Cartan name X_n (or $I_2(n)$ if $X =$`"I"`).

If the `Symmetric` flag is set `true`, the symmetric Cartan matrix with $c_{ij} = c_{ji} = -2\cos(\pi/m_{ij})$ is returned.

The `BaseField` flag determines which field the Cartan matrix is defined over. If the matrix is crystallographic however, it is defined over the integers regardless of the value of this flag. The possible values are:

1. `"NumberField"`: An algebraic number field. This is the default. See Chapter 35.
2. `"Cyclotomic"` or `"SparseCyclotomic"`: A cyclotomic field with the sparse representation for elements. See Chapter 37.
3. `"DenseCyclotomic"`: A cyclotomic field with the dense representation for elements. See Chapter 37.

`IrreducibleDynkinDigraph(X, n)`

The irreducible Dynkin digraph with Cartan name X_n. The Cartan name must be crystallographic, i.e. it cannot involve types H_3, H_4 or $I_2(m)$.

Example H101E13

These functions are useful in loops.

```
> for n in [1..5] do
>     IsTree(IrreducibleCoxeterGraph("A~", n));
> end for;
true
false
false
false
false
> C := &join[ IrreducibleCoxeterGraph(t, 4) : t in ["A","B","C","D","F"] ];
```

`IsCoxeterIsomorphic(N1, N2)`

Returns `true` if and only if the Cartan names given by the strings N_1 and N_2 correspond to isomorphic Coxeter systems.

`IsCartanEquivalent(N1, N2)`

Returns `true` if and only if the Cartan names given by the strings N_1 and N_2 correspond to Cartan equivalent Cartan matrices. The Cartan names must be crystallographic; i.e., they cannot involve types H_3, H_4 and $I_2(m)$.

Example H101E14

```
> IsCoxeterIsomorphic("A1A1", "D2");
true
> IsCoxeterIsomorphic("B5", "C5");
true
> IsCartanEquivalent("B5", "C5");
false
```

`IsSimplyLaced(N)`

Returns `true` if, and only if, the Coxeter matrix with Cartan name given by the string N is simply laced, i.e. all its entries are 1, 2, or 3.

`CoxeterGroupOrder(N)`

`CoxeterGroupFactoredOrder(N)`

The (factored) order of the Coxeter group with Cartan name given by the string N.

`NumberOfPositiveRoots(N)`

`NumPosRoots(N)`

The number of positive roots of the Coxeter group with Cartan name given by the string N. See Subsection 102.1.3 for the definition of positive roots.

`FundamentalGroup(N)`

The fundamental group of the crystallographic Cartan matrix with Cartan name given by the string N, i.e. $\mathbf{Z}^n/\Gamma$ where Γ is the lattice generated by the rows of the Cartan matrix. The natural mapping $\mathbf{Z}^n \to \mathbf{Z}^n/\Gamma$ is the second returned value.

Example H101E15

```
> CoxeterGroupOrder("F4");
1152
> CoxeterGroupFactoredOrder("F4");
[ <2, 7>, <3, 2> ]
> NumPosRoots("F4");
24
> #FundamentalGroup("F4");
1
```

`CartanName(M)`

`CartanName(G)`

`CartanName(C)`

`CartanName(D)`

The Cartan name of a Coxeter matrix M, Coxeter graph G, Cartan matrix C, or Dynkin digraph D. If the corresponding Coxeter group is neither finite nor affine, an error is flagged.

Example H101E16

```
> CartanName(SymmetricMatrix([1, 3,1, 2,3,1]));
A3
> CartanName(SymmetricMatrix([1, 3,1, 3,3,1]));
A~2
> CartanName(SymmetricMatrix([1, 3,1, 4,3,1]));
The component at rows and columns [ 1, 2, 3 ]
is not a finite or affine Coxeter matrix
> C := Matrix(4,4, [2,-2,0,0, -1,2,0,0, 0,0,2,-2, 0,0,-1,2] );
> C;
[ 2 -2  0  0]
[-1  2  0  0]
[ 0  0  2 -2]
```

```
[ 0  0 -1  2]
```

DynkinDiagram(M)
DynkinDiagram(G)
DynkinDiagram(C)
DynkinDiagram(D)
DynkinDiagram(N)

Print the Dynkin diagram of a Coxeter matrix M, Coxeter graph G, Cartan matrix C, Dynkin digraph D or Cartan name given by the string N. If the corresponding group is neither affine nor crystallographic, an error is flagged.

Example H101E17

```
> DynkinDiagram("A~5 D4 BC3");
A~5    1 - 2 - 3 - 4 - 5
       |               |
       ------- 6 -------
D4     9
      /
7 - 8
      \
       10
BC3    11 - 12 =>= 13
```

CoxeterDiagram(M)
CoxeterDiagram(G)
CoxeterDiagram(C)
CoxeterDiagram(D)
CoxeterDiagram(N)

Print the Coxeter diagram of a Coxeter matrix M, Coxeter graph G, Cartan matrix C, Dynkin digraph D or Cartan name given by the string N. If the corresponding group is not affine or is not crystallographic, an error is flagged.

Example H101E18

```
> CoxeterDiagram("A~5 D4 BC3");
A~5    1 - 2 - 3 - 4 - 5
       |               |
       ------- 6 -------
D4     9
      /
7 - 8
      \
       10
BC3    11 - 12 === 13
```

101.7 Hyperbolic Groups

A hyperbolic reflection group is a group generated by reflections in hyperbolic space. A Coxeter group is called *hyperbolic* if it is infinite, nonaffine, and it has a representation as a discrete, properly acting, hyperbolic reflection group whose Tits' cone consists entirely of vectors with negative norm (see [Bou68] for more details). A hyperbolic reflection group is *compact hyperbolic* if it is hyperbolic with a compact fundamental region.

Every infinite nonaffine Coxeter group of rank 3 is hyperbolic. There are only 72 hyperbolic groups of rank larger than 3 which, for convenience, are numbered from 1 to 72. The numbering is essentially arbitrary.

`IsCoxeterHyperbolic(M)`

`IsCoxeterCompactHyperbolic(M)`

Returns `true` if, and only if, the matrix M is the Coxeter matrix of a (compact) hyperbolic Coxeter group.

`IsCoxeterHyperbolic(G)`

`IsCoxeterCompactHyperbolic(G)`

Returns `true` if, and only if, the graph G is the Coxeter graph of a (compact) hyperbolic Coxeter group.

`HyperbolicCoxeterMatrix(i)`

The Coxeter matrix of the ith hyperbolic Coxeter group of rank larger than 3.

`HyperbolicCoxeterGraph(i)`

The Coxeter graph of the ith hyperbolic Coxeter group of rank larger than 3.

Example H101E19

```
> for i in [1..72] do
>   if IsCoxeterCompactHyperbolic(HyperbolicCoxeterMatrix(i)) then
>     printf "%o, ", i;
>   end if;
> end for;
1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14,
```

101.8 Related Structures

In this section functions for creating other structures from Coxeter matrices, Coxeter graphs, Cartan matrices, Dynkin diagrams, and Cartan names are listed. The reader is referred to the appropriate sections of the Handbook for more details.

`RootSystem(M)`
`RootSystem(G)`
`RootSystem(C)`
`RootSystem(D)`
`RootSystem(N)`

The finite root system of a Coxeter matrix M, Coxeter graph G, Cartan matrix C, Dynkin digraph D, or Cartan name given by the string N. If the corresponding Coxeter group is infinite, an error is flagged. See Chapter 102.

`RootDatum(C)`
`RootDatum(M)`
`RootDatum(G)`
`RootDatum(D)`
`RootDatum(N)`

The finite root datum of a crystallographic Cartan matrix C, Coxeter matrix M, Coxeter graph G, Dynkin digraph D, or Cartan name given by the string N. If the corresponding Coxeter group is infinite, an error is flagged. See Chapter 103.

```
CoxeterGroup(GrpFPCox, M)
CoxeterGroup(GrpFPCox, G)
CoxeterGroup(GrpFPCox, C)
CoxeterGroup(GrpFPCox, D)
CoxeterGroup(GrpFPCox, N)
```

The Coxeter group of a Coxeter matrix M, Coxeter graph G, Cartan matrix C, Dynkin digraph D, or Cartan name given by the string N. See Chapter 104.

```
CoxeterGroup(GrpPermCox, M)
CoxeterGroup(GrpPermCox, G)
CoxeterGroup(GrpPermCox, C)
CoxeterGroup(GrpPermCox, D)
CoxeterGroup(GrpPermCox, N)
```

The permutation Coxeter group of a Coxeter matrix M, Coxeter graph G, Cartan matrix C, Dynkin digraph D, or Cartan name given by the string N. If the corresponding Coxeter group is infinite, an error is flagged. See Chapter 104.

```
CoxeterGroup(M)
CoxeterGroup(G)
CoxeterGroup(C)
CoxeterGroup(D)
CoxeterGroup(N)
```

The Coxeter group of a Coxeter matrix M, Coxeter graph G, Cartan matrix C, Dynkin digraph D, or Cartan name given by the string N. If the corresponding Coxeter group is finite, it is returned as a permutation group; otherwise it is returned as a finitely presented group.

```
ReflectionGroup(M)
ReflectionGroup(G)
ReflectionGroup(C)
ReflectionGroup(D)
ReflectionGroup(N)
```

The reflection group of a Coxeter matrix M, Coxeter graph G, Cartan matrix C, Dynkin digraph D, or Cartan name given by the string N. See Chapter 105.

`LieAlgebra(C, k)`

`LieAlgebra(D, k)`

`LieAlgebra(N, k)`

The Lie algebra over the ring k of a crystallographic Cartan matrix C, Dynkin digraph D, or Cartan name given by the string N. If the corresponding Coxeter group is infinite, an error is flagged. See Chapter 106.

`MatrixLieAlgebra(C, k)`

`MatrixLieAlgebra(D, k)`

`MatrixLieAlgebra(N, k)`

The Lie algebra over the ring k of a crystallographic Cartan matrix C, Dynkin digraph D, or Cartan name given by the string N. If the corresponding Coxeter group is infinite, an error is flagged. See Chapter 106.

`GroupOfLieType(C, k)`

`GroupOfLieType(D, k)`

`GroupOfLieType(N, k)`

The group of Lie type over the ring k of a crystallographic Cartan matrix C, Dynkin digraph D, or Cartan name given by the string N. If the corresponding Coxeter group is infinite, an error is flagged. See Chapter 109.

101.9 Bibliography

[**Bou68**] N. Bourbaki. *Éléments de mathématique. Fasc. XXXIV. Groupes et algèbres de Lie. Chapitre IV: Groupes de Coxeter et systèmes de Tits. Chapitre V: Groupes engendrés par des réflexions. Chapitre VI: Systèmes de racines.* Hermann, Paris, 1968.

[**Car52**] Elie Cartan. *Œuvres complètes. Partie I. Groupes de Lie.* Gauthier-Villars, Paris, 1952.

[**Cox34**] H. S. M. Coxeter. Discrete groups generated by reflections. *Ann. of Math.*, 35:588–621, 1934.

102 ROOT SYSTEMS

102.1 Introduction 3065

102.1.1 Reflections 3065

102.1.2 Definition of a Root System . . 3065

102.1.3 Simple and Positive Roots . . . 3066

102.1.4 The Coxeter Group 3066

102.1.5 Nonreduced Root Systems . . . 3067

102.2 Constructing Root Systems . 3067

RootSystem(N) 3068
RootSystem(M) 3068
RootSystem(G) 3068
RootSystem(C) 3068
RootSystem(D) 3069
RootSystem(A, B) 3069
IrreducibleRootSystem(X, n) 3070
StandardRootSystem(X, n) 3070
ToralRootSystem(n) 3070
TrivialRootSystem() 3070

102.3 Operators on Root Systems . 3071

eq 3071
IsIsomorphic(R1, R2) 3071
IsCartanEquivalent(R1, R2) 3071
CartanName(R) 3071
CoxeterDiagram(R) 3071
DynkinDiagram(R) 3071
CoxeterMatrix(R) 3071
CoxeterGraph(R) 3071
CartanMatrix(R) 3071
DynkinDigraph(R) 3071
BaseField(R) 3072
BaseRing(R) 3072
RealInjection(R) 3072
Rank(R) 3072
Dimension(R) 3072
CoxeterGroupOrder(R) 3072

102.4 Properties of Root Systems . 3073

IsIrreducible(R) 3073
IsProjectivelyIrreducible(R) 3073
IsReduced(R) 3073
IsSemisimple(R) 3073
IsCrystallographic(R) 3073
IsSimplyLaced(R) 3073

102.5 Roots and Coroots 3074

102.5.1 Accessing Roots and Coroots . . 3074

RootSpace(R) 3074
CorootSpace(R) 3074
SimpleRoots(R) 3074
SimpleCoroots(R) 3074
NumberOfPositiveRoots(R) 3075
NumPosRoots(R) 3075
Roots(R) 3075
Coroots(R) 3075
PositiveRoots(R) 3075
PositiveCoroots(R) 3075
Root(R, r) 3075
Coroot(R, r) 3075
RootPosition(R, v) 3075
CorootPosition(R, v) 3075
HighestRoot(R) 3076
HighestCoroot(R) 3076
HighestLongRoot(R) 3076
HighestLongCoroot(R) 3076
HighestShortRoot(R) 3076
HighestShortCoroot(R) 3076
CoxeterForm(R) 3077
DualCoxeterForm(R) 3077

102.5.2 Reflections 3077

SimpleReflectionMatrices(R) 3077
SimpleCoreflectionMatrices(R) 3077
ReflectionMatrices(R) 3077
CoreflectionMatrices(R) 3077
ReflectionMatrix(R, r) 3077
CoreflectionMatrix(R, r) 3077
SimpleReflectionPermutations(R) 3078
ReflectionPermutations(R) 3078
ReflectionPermutation(R, r) 3078
ReflectionWords(R) 3078
ReflectionWord(R, r) 3078

102.5.3 Operations and Properties for Roots and Coroot Indices . . . 3079

Sum(R, r, s) 3079
IsPositive(R, r) 3079
IsNegative(R, r) 3079
Negative(R, r) 3079
RootHeight(R, r) 3079
CorootHeight(R, r) 3079
RootNorms(R) 3079
CorootNorms(R) 3079
RootNorm(R, r) 3080
CorootNorm(R, r) 3080
IsLongRoot(R, r) 3080
IsShortRoot(R, r) 3080
IsIndivisibleRoot(R, r) 3080
LeftString(R, r, s) 3080
RightString(R, r, s) 3080
LeftStringLength(R, r, s) 3080
RightStringLength(R, r, s) 3080
AdditiveOrder(R) 3081
IsAdditiveOrder(R, Q) 3081

102.6 Building Root Systems . . . 3082

sub< > 3082
sub< > 3082

subset 3082
+ 3082
DirectSum(R1, R2) 3082
join 3082
DirectSumDecomposition(R) 3083
IndecomposableSummands(R) 3083
Dual(R) 3083
IndivisibleSubsystem(R) 3083

102.7 Related Structures 3084

RootDatum(R) 3084
CoxeterGroup(GrpFPCox, R) 3084
CoxeterGroup(R) 3084
CoxeterGroup(GrpPermCox, R) 3084
ReflectionGroup(R) 3084
CoxeterGroup(GrpMat, W) 3084
LieAlgebra(R, k) 3084
MatrixLieAlgebra(R, k) 3084

102.8 Bibliography 3084

Chapter 102
ROOT SYSTEMS

102.1 Introduction

This chapter describes MAGMA functions for computing with finite real root systems. A root system describes the reflections in a reflection group (Chapter 105). Root systems are essential in the theories of finite Coxeter groups (Chapter 104) and Lie algebras (Chapter 106). See [Bou68] for more details on the theory of root systems. The closely related concept of a root datum is discussed in Chapter 103.

102.1.1 Reflections

Let X and Y be vector spaces over a field k with bilinear pairing $\langle \circ, \circ \rangle : X \times Y \to k$ that identifies Y with the dual of X. Given nonzero $\alpha \in X$ and $\alpha^\star \in Y$, the linear map $s_\alpha : X \to X$ is defined by

$$x s_\alpha = x - \langle x, \alpha^\star \rangle \alpha$$

and the linear map $s_\alpha^\star : Y \to Y$ by

$$y s_\alpha^\star = y - \langle \alpha, y \rangle \alpha^\star.$$

These maps are called *reflections* if one of the following equivalent properties hold: $\langle \alpha, \alpha^\star \rangle = 2$; ${s_\alpha}^2 = 1$; $\langle x s_\alpha, y s_\alpha^\star \rangle = \langle x, y \rangle$ for all $x \in X$ and $y \in Y$; $\alpha s_\alpha = -\alpha$. The mapping $s_\alpha^\star$ is also called a *coreflection*: this just means it is a reflection defined on Y instead of X. MAGMA functions for computing with reflections are described in Section 105.2.

If X has an inner product, then we can take $Y = X$ and use the inner product as our pairing. In MAGMA, we generally take $X = Y$ to be a row space, with the bilinear pairing given by the standard inner product $\langle x, y \rangle = xy^T$. However, it is sometimes useful to allow X and Y to be distinct subspaces of a row space.

For the purposes of this chapter, k will always be the rational field (Chapter 20), a number field (Chapter 20), or a cyclotomic field (Chapter 37). The real field (Chapter 25) is *not* allowed since it is not infinite precision.

102.1.2 Definition of a Root System

Suppose Φ is a finite subset of $X \setminus \{0\}$. For each α in Φ, suppose a corresponding nonzero $\alpha^\star$ in Y is given; set $\Phi^\star = \{\alpha^\star \mid \alpha \in \Phi\}$. The tuple $R = (X, \Phi, Y, \Phi^\star)$ is called a *root system* if the following conditions are satisfied for every α in Φ

1. s_α and $s_\alpha^\star$ are reflections;
2. Φ is closed under the action of s_α; and
3. $\Phi^\star$ is closed under the action of $s_\alpha^\star$.

The set X is called the *root space* and Y is called the *coroot space*. The elements of Φ are called *roots* and the elements of $\Phi^\star$ are called *coroots*. A root system is said to be *crystallographic* if $\langle \alpha, \beta^\star \rangle$ is integral for every root α and coroot $\beta^\star$. A root system is *reduced*, if $\alpha, \beta \in \Phi$ with β a scalar product of α implies $\alpha = \pm\beta$. Note that it is possible for the set of roots to be empty, in which case the system is called *toral*.

102.1.3 Simple and Positive Roots

A subset Δ of Φ is called a set of *simple roots* if

1. Δ is a basis for the span of the roots $k\Phi \leq X$; and
2. $\Phi = \Phi^+ \cup \Phi^-$, where Φ^+ is the set of linear combinations of elements of Δ with nonnegative coefficients, and $\Phi^- = -\Phi^+$.

Every root system has a set of simple roots. Simple roots are frequently called fundamental roots. The elements of Φ^+ are called *positive roots* and the elements of Φ^- are called *negative roots*. The coroots corresponding to the simple (respectively, positive, negative) roots are the *simple* (respectively, *positive*, *negative*) *coroots*.

The *rank* of a root system is the size of Δ, i.e. the dimension of the subspace $k\Phi$. The rank cannot be larger than the *dimension* of the root system (i.e. the dimension of X); if the rank and dimension are equal, the root system is said to be *semisimple*.

Choose a basis $e_1, \ldots, e_d$ for X and a dual basis $f_1, \ldots, f_d$ for Y, so that $\langle e_i, f_j \rangle = \delta_{ij}$. A reduced root system is determined by a pair of real matrices A and B where the rows of A are the simple roots and the rows of B are the corresponding coroots; i.e. $A_{ij} = \langle \alpha_i, f_j \rangle$ and $B_{ij} = \langle e_j, \alpha_i^\star \rangle$.

102.1.4 The Coxeter Group

The group W generated by the reflections s_α, for α a simple root, is a finite Coxeter group. The *Cartan matrix* of a root system is

$$C = \left(\left\langle\, \alpha_i, \alpha_j^\star \,\right\rangle \right)_{i,j=1}^{n} = AB^t.$$

Note that the root system is crystallographic if, and only if, its Cartan matrix is crystallographic. As in Chapter 101, the Cartan matrix is used to define the Coxeter matrix, Coxeter graph, and Dynkin digraph of a root system.

The classification of Section 101.6 applies to reduced semisimple root systems. The isomorphism class of a reduced root system is determined by its Coxeter graph and its dimension.

A *Coxeter form* is a W-invariant bilinear form on X. If R is reduced and irreducible, then the roots can have at most two different lengths with respect to this form. We call the roots *long* or *short* accordingly. The Coxeter form is normalised so that the short roots in each component have length one. Note that, even if $X = Y$, this form will generally not be the same as the pairing $\langle \circ, \circ \rangle$; however it can be arranged for them to be the same (see `StandardRootSystem`).

102.1.5 Nonreduced Root Systems

A root system is *reduced*, if $\alpha, \beta \in \Phi$ with β a scalar product of α implies $\alpha = \pm\beta$. A root α with the property $2\alpha \notin \Phi$ is called *reduced*. A root α with the property $\frac{1}{2}\alpha \in \Phi$ is called *divisible*. If R is a root system, then the set R_0 of indivisible roots in R form the *indivisible subsystem*.

Let R be a nonreduced irreducible *crystallographic* root system of rank n. It can be shown that R_0 is irreducible of type of type B_n and every root is either in R_0, or is two times a short root of R_0. The Cartan type of R in this case is BC_n. For noncrystallographic root systems the situation is more complex.

Note that the Cartan matrix, Coxeter matrix, Coxeter diagram, Coxeter group and Dynkin diagram are the same for R and R_0. Thus, when creating a non-reduced crystallographic root system for a given Cartan matrix, Coxeter matrix, Coxeter diagram, Coxeter group or Dynkin diagram, one must specify the set of nonreduced simple roots. For example, let C be a cartan matrix of type $B_2 \times B_3$. Then the set of non-reduced fundamental roots can be one of $\emptyset$, $\{2\}$, $\{5\}$, or $\{2, 5\}$, in which cases the root system will be of types $B_2 \times B_3$, $BC_2 \times B_3$, $B_2 \times BC_3$, or $BC_2 \times BC_3$ respectively.

102.2 Constructing Root Systems

We first describe some optional parameters that are common to many functions described in this section.

`RealInjection` ANY *Default :* `false`

Number field elements and cyclotomic field elements do not have a natural identification with real numbers. The `RealInjection` flag allows the user to provide one. If the base field of the Cartan matrix C is a number field, the flag should be an injection into the real field; if the base field is cyclotomic, the flag should be an injection into the complex field taking real values on the entries of C (see more in Section 101.4).

`Nonreduced` SETENUM *Default :* {}

The optional argument `Nonreduced` is used to distinguish the reducedness of a root system in case the input doesn't uniquely determine it.

`Symmetric` BOOLELT *Default :* `false`

If the `Symmetric` flag is set `true`, the symmetric Cartan matrix is used. For types $I_2(m)$, H_3, H_4 the symmetric Cartan matrix is *always* used, since the root system is nonreduced otherwise.

`BaseField` MONSTGELT *Default :* "*NumberField*"

The `BaseField` flag determines the field over which the Cartan matrix is defined. The possible values are:

1. `"NumberField"`: An algebraic number field. This is the default. See Chapter 35.
2. `"Cyclotomic"` or `"SparseCyclotomic"`: A cyclotomic field with the sparse representation for elements. See Chapter 37.

3. `"DenseCyclotomic"`: A cyclotomic field with the dense representation for elements. See Chapter 37.

`RootSystem(N)`

`Symmetric`	BoolElt	*Default :* `false`
`BaseField`	MonStgElt	*Default :* "*NumberField*"

The root system with Cartan name given by the string N. In addition to the Cartan names in Section 101.6, we allow `"BCn"` for the irreducible nonreduced system, and `"Tn"` for the n-dimensional toral subsystem. Note that `"Tn"` is used for input only and does not appear in the string returned by `CartanName` when applied to the resulting root system (see example below). For descriptions of the parameters `Symmetric` and `BaseField` see the beginning of this section

Example H102E1

```
> RootSystem("H3 E6");
Root system of type H3 E6
> RootSystem("A2 T1 I2(5)");
Root system of type A2 I2(5)
```

`RootSystem(M)`

`RootSystem(G)`

`Nonreduced`	SetEnum	*Default :* {}
`Symmetric`	BoolElt	*Default :* `false`
`BaseField`	MonStgElt	*Default :* "*NumberField*"

The semisimple root system with Coxeter matrix M or Coxeter graph G (see Chapter 101). If the corresponding Coxeter group is infinite, an error is flagged. For descriptions of the parameters `Nonreduced`, `Symmetric`, and `BaseField` see the beginning of this section.

`RootSystem(C)`

`RealInjection`	Any	*Default :* `false`
`Nonreduced`	SetEnum	*Default :* {}

The semisimple root system with Cartan matrix C (see Chapter 101). If the corresponding Coxeter group is infinite, an error is flagged. For descriptions of the parameters `RealInjection` and `Nonreduced` see the beginning of this section.

`RootSystem(D)`

`Nonreduced` SetEnum *Default :* {}

The semisimple crystallographic root system with Cartan matrix C, or Dynkin diagram D (see Chapter 101). If the corresponding Coxeter group is infinite, an error is flagged. For a description of the parameter `Nonreduced` see the beginning of this section.

Example H102E2

```
> M := SymmetricMatrix([1, 3,1, 2,3,1]);
> RootSystem(M);
Root system of type A3
> M := SymmetricMatrix([1, 3,1, 3,3,1]);
> RootSystem(M);
>> RootSystem(M);
             ^
Runtime error in 'RootSystem': Not a finite root system in rows/columns
[ 1, 2, 3 ]
```

`RootSystem(A, B)`

`RealInjection` Any *Default :* `false`

`Nonreduced` SetEnum *Default :* {}

The root system with simple roots given by the rows of the matrix A and simple coroots given by the rows of the matrix B. The matrices A and B must have the following properties:

1. A and B must have the same number of rows and the same number of columns; they must be defined over the same ring, which must be the integers, the rational field, a number field, or a cyclotomic field;
2. the number of columns must be at least the number of rows; and
3. AB^t must be the Cartan matrix of a finite Coxeter group.

For descriptions of the parameters `RealInjection` and `Nonreduced` see the beginning of this section.

Example H102E3

The following code creates a nonsemisimple root system of type G_2.

```
> A := Matrix(2,3, [1,-1,0, -1,1,-1]);
> B := Matrix(2,3, [1,-1,1, 0,1,-1]);
> RootSystem(A, B);
Root system of type G2
```

`IrreducibleRootSystem(X, n)`

`Symmetric`	BoolElt	*Default :* `false`
`BaseField`	MonStgElt	*Default :* "*NumberField*"

The irreducible root system with Cartan name X_n (or $I_2(n)$ if $X =$ "I") given by the string X and integer n. In addition to the Cartan names in Section 101.6, we allow `"BCn"` for the irreducible nonreduced system. For descriptions of the parameters `Symmetric` and `BaseField` see the beginning of this section.

`StandardRootSystem(X, n)`

The standard root system with Cartan name X_n (or $I_2(n)$ if $X =$ "I") given by the string X and integer n, i.e. the root system whose Coxeter form is the same as the standard inner product. In addition to the Cartan names in Section 101.6, we allow `"BCn"` for the irreducible nonreduced system. For type A_n, the standard root system is not semisimple.

Example H102E4

```
> Rs := { IrreducibleRootSystem("I", n) : n in [3..20] };
> { R : R in Rs | IsCrystallographic(R) };
{
    Root system of type I2(3) ,
    Root system of type I2(4) ,
    Root system of type I2(6)
}
```

`ToralRootSystem(n)`

The toral root system of dimension n, i.e., the n-dimensional root system with no roots or coroots.

`TrivialRootSystem()`

The trivial root system of dimension 0.

102.3 Operators on Root Systems

`R1 eq R2`

Returns `true` if, and only if, the root systems R_1 and R_2 are identical.

`IsIsomorphic(R1, R2)`

Returns `true` if, and only if, root systems R_1 and R_2 are isomorphic.

`IsCartanEquivalent(R1, R2)`

Returns `true` if, and only if, the crystallographic root systems R_1 and R_2 are Cartan equivalent, i.e. their Cartan matrices are the same modulo a permutation of the underlying basis.

Example H102E5

Note that the root systems B_n and C_n are isomorphic but not Cartan equivalent. Hence Cartan equivalence is *not* an invariant of a root system since it depends on the particular representation of the (co)roots within the (co)root space.

```
> R := RootSystem("B4");  S := RootSystem("C4");
> IsIsomorphic(R, S);
true
> IsCartanEquivalent(R, S);
false
```

`CartanName(R)`

The Cartan name of the root system R (Section 101.6).

`CoxeterDiagram(R)`

Print the Coxeter diagram of the root system R (Section 101.6).

`DynkinDiagram(R)`

Print the Dynkin diagram of the root system R (Section 101.6). If R is not crystallographic, an error is flagged.

`CoxeterMatrix(R)`

The Coxeter matrix of the root system R (Section 101.2).

`CoxeterGraph(R)`

The Coxeter graph of the root system R (Section 101.3).

`CartanMatrix(R)`

The Cartan matrix of the root system R (Section 101.4).

`DynkinDigraph(R)`

The Dynkin digraph of the root system R (Section 101.5). If R is not crystallographic, an error is flagged.

Example H102E6

```
> R := RootSystem("F4");
> DynkinDiagram(R);
F4    1 - 2 =>= 3 - 4
> CoxeterDiagram(R);
F4    1 - 2 === 3 - 4
```

`BaseField(R)`
`BaseRing(R)`

The field over which the root system R is defined.

`RealInjection(R)`

The real injection of the root system R (Section 102.2).

`Rank(R)`

The rank of the root system R, i.e. the number of simple (co)roots.

`Dimension(R)`

The dimension of the root system R, i.e. the dimension of the (co)root space. This is always at least as large as the rank, with equality when R is semisimple.

`CoxeterGroupOrder(R)`

The order of the Coxeter group of the root system R.

Example H102E7

```
> R := RootSystem("I2(7)");
> BaseField(R);
Number Field with defining polynomial x^3 - x^2 - 2*x + 1 over the
Rational Field
> Rank(R) eq Dimension(R);
true
> CoxeterGroupOrder(R);
14
```

102.4 Properties of Root Systems

`IsIrreducible(R)`

Returns `true` if, and only if, the root system R is irreducible.

`IsProjectivelyIrreducible(R)`

Returns `true` if, and only if, the root system R is a direct sum of a simple system and a toral system. This is equivalent to R having a connected Coxeter diagram.

`IsReduced(R)`

Returns `true` if, and only if, the root system R is reduced.

`IsSemisimple(R)`

Returns `true` if, and only if, the root system R is semisimple, i.e. its rank is equal to its dimension.

`IsCrystallographic(R)`

Returns `true` if, and only if, the root system R is crystallographic, i.e. its Cartan matrix is integral.

`IsSimplyLaced(R)`

Returns `true` if, and only if, the root system R is simply laced, i.e. its Coxeter graph contains no labelled edges.

Example H102E8

```
> R := RootSystem("A5 B2");
> IsIrreducible(R);
false
> IsSemisimple(R);
true
> IsCrystallographic(R);
true
> IsSimplyLaced(R);
false
```

102.5 Roots and Coroots

The roots are stored as an indexed set

$$\{@\ \alpha_1, \ldots, \alpha_N, \alpha_{N+1}, \ldots, \alpha_{2N}\ @\},$$

where $\alpha_1, \ldots, \alpha_N$ are the positive roots (in an order compatible with height), and $\alpha_{N+1}, \ldots, \alpha_{2N}$ are the corresponding negative roots (i.e. $\alpha_{i+N} = -\alpha_i$). The simple roots are $\alpha_1, \ldots, \alpha_n$ where n is the rank.

Many of these functions have an optional argument `Basis` which may take one of the following values

1. `"Standard"`: the standard basis for the (co)root space (this is the default); or
2. `"Root"`: the basis of simple (co)roots.

102.5.1 Accessing Roots and Coroots

`RootSpace(R)`

`CorootSpace(R)`

The vector space containing the (co)roots of the root system R, i.e. X (respectively, Y).

`SimpleRoots(R)`

`SimpleCoroots(R)`

The simple (co)roots of the root system R as the rows of a matrix, i.e. A (respectively, B).

Example H102E9

```
> R := RootSystem("G2");
> RootSpace(R);
Full Vector space of degree 2 over Rational Field
> CorootSpace(R);
Full Vector space of degree 2 over Rational Field
> SimpleRoots(R);
[1 0]
[0 1]
> SimpleCoroots(R);
[ 2 -3]
[-1  2]
> CartanMatrix(R);
[ 2 -1]
[-3  2]
```

`NumberOfPositiveRoots(R)`

`NumPosRoots(R)`

The number of positive roots of the root system R. This is also the number of positive coroots. The total number of (co)roots is twice the number of positive (co)roots.

`Roots(R)`

`Coroots(R)`

`Basis` MonStgElt *Default :* "*Standard*"

The indexed set of (co)roots of the root system R, i.e. $\{@ \alpha_1, \ldots \alpha_{2N} @\}$ (respectively, $\{@ \alpha_1^\star, \ldots \alpha_{2N}^\star @\}$).

`PositiveRoots(R)`

`PositiveCoroots(R)`

`Basis` MonStgElt *Default :* "*Standard*"

The indexed set of positive (co)roots of the root system R, i.e. $\{@ \alpha_1, \ldots \alpha_N @\}$ (respectively, $\{@ \alpha_1^\star, \ldots \alpha_N^\star @\}$).

`Root(R, r)`

`Coroot(R, r)`

`Basis` MonStgElt *Default :* "*Standard*"

The rth (co)root α_r (respectively, $\alpha_r^\star$) of the root system R.

`RootPosition(R, v)`

`CorootPosition(R, v)`

`Basis` MonStgElt *Default :* "*Standard*"

If v is a (co)root in the root system R, return its index; otherwise return 0. These functions will try to coerce v, which can be a vector or a sequence representing a vector, into the appropriate vector space; v should be written with respect to the basis specified by the parameter `Basis`.

Example H102E10

```
> A := Matrix(2,3, [1,-1,0, -1,1,-1]);
> B := Matrix(2,3, [1,-1,1, 0,1,-1]);
> R := RootSystem(A, B);
> Roots(R);
{@
    (1 -1  0),
    (-1  1 -1),
    (0  0 -1),
    (1 -1 -1),
```

```
    (2 -2 -1),
    (1 -1 -2),
    (-1  1  0),
    (1 -1  1),
    (0 0 1),
    (-1  1  1),
    (-2  2  1),
    (-1  1  2)
@}
> PositiveCoroots(R);
{@
    (1 -1  1),
    (0  1 -1),
    (1  2 -2),
    (2  1 -1),
    (1 0 0),
    (1  1 -1)
@}
> #Roots(R) eq 2*NumPosRoots(R);
true
> Root(R, 4);
(1 -1 -1)
> Root(R, 4 : Basis := "Root");
(2 1)
> RootPosition(R, [1,-1,-1]);
4
> RootPosition(R, [2,1] : Basis := "Root");
4
```

`HighestRoot(R)`

`HighestCoroot(R)`

`Basis` MonStgElt *Default :* "*Standard*"

The unique (co)root of greatest height in the irreducible root system R.

`HighestLongRoot(R)`

`HighestLongCoroot(R)`

`Basis` MonStgElt *Default :* "*Standard*"

The unique long (co)root of greatest height in the irreducible root system R.

`HighestShortRoot(R)`

`HighestShortCoroot(R)`

`Basis` MonStgElt *Default :* "*Standard*"

The unique short (co)root of greatest height in the irreducible root system R.

Example H102E11

```
> R := RootSystem("G2");
> HighestRoot(R);
(3 2)
> HighestLongRoot(R);
(3 2)
> HighestShortRoot(R);
(2 1)
```

`CoxeterForm(R)`

`DualCoxeterForm(R)`

`Basis` MonStgElt *Default :* "*Standard*"

The matrix of an inner product on the (co)root space of the root system R which is invariant under the action of the (co)roots. This inner product is uniquely determined up to a constant on each irreducible component of R. The inner product is normalised so that the short roots in each crystallographic component have length one.

102.5.2 Reflections

The root α acts on the root space via the reflection s_α; the coroot $\alpha^\star$ acts on the coroot space via the coreflection $s^\star_\alpha$.

`SimpleReflectionMatrices(R)`

`SimpleCoreflectionMatrices(R)`

`Basis` MonStgElt *Default :* "*Standard*"

The sequence of matrices giving the action of the simple (co)roots of the root system R on the (co)root space, i.e. the matrices of $s_{\alpha_1}, \ldots, s_{\alpha_n}$ (respectively, $s^\star_{\alpha_1}, \ldots, s^\star_{\alpha_n}$).

`ReflectionMatrices(R)`

`CoreflectionMatrices(R)`

`Basis` MonStgElt *Default :* "*Standard*"

The sequence of matrices giving the action of the (co)roots of the root system R on the (co)root space, i.e. the matrices of $s_{\alpha_1}, \ldots, s_{\alpha_{2N}}$ (respectively, $s^\star_{\alpha_1}, \ldots, s^\star_{\alpha_{2N}}$).

`ReflectionMatrix(R, r)`

`CoreflectionMatrix(R, r)`

`Basis` MonStgElt *Default :* "*Standard*"

The matrix giving the action of the rth (co)root of the root system R on the (co)root space, i.e. the matrix of s_{α_r} (respectively, $s^\star_{\alpha_r}$).

`SimpleReflectionPermutations(R)`

The sequence of permutations giving the action of the simple (co)roots of the root system R on the (co)roots. This action is the same for roots and coroots.

`ReflectionPermutations(R)`

The sequence of permutations giving the action of the (co)roots of the root system R on the (co)roots. This action is the same for roots and coroots.

`ReflectionPermutation(R, r)`

The permutation giving the action of the rth (co)root of the root system R on the (co)roots. This action is the same for roots and coroots.

`ReflectionWords(R)`

The sequence of words in the simple reflections for all the reflections of the root system R. These words are given as sequences of integers. In other words, if $[a_1, \ldots, a_l] =$ `ReflectionWords(R)[r]`, then $s_{\alpha_r} = s_{\alpha_{a_1}} \cdots s_{\alpha_{a_l}}$.

`ReflectionWord(R, r)`

The word in the simple reflections for the rth reflection of the root system R. The word is given as a sequence of integers. In other words, if $[a_1, \ldots, a_l] =$ `ReflectionWord(R,r)`, then $s_{\alpha_r} = s_{\alpha_{a_1}} \cdots s_{\alpha_{a_l}}$.

Example H102E12

```
> R := RootSystem("B3");
> mx := ReflectionMatrix(R, 4);
> perm := ReflectionPermutation(R, 4);
> wd := ReflectionWord(R, 4);
> RootPosition(R, Root(R,2) * mx) eq 2^perm;
true
> perm eq &*[ ReflectionPermutation(R, r) : r in wd ];
true
>
> mx := CoreflectionMatrix(R, 4);
> CorootPosition(R, Coroot(R,2) * mx) eq 2^perm;
true
```

102.5.3 Operations and Properties for Roots and Coroot Indices

`Sum(R, r, s)`

The index of the sum of the rth and sth roots in the crystallographic root system R, or 0 if the sum is not a root. In other words, if $t =$ `Sum(R,`r`,`s`)` $\neq 0$ then $\alpha_t = \alpha_r + \alpha_s$. We require $\alpha_r \neq \pm\alpha_s$.

`IsPositive(R, r)`

Returns `true` if, and only if, the rth (co)root of the root system R is a positive root.

`IsNegative(R, r)`

Returns `true` if, and only if, the rth (co)root of the root system R is a negative root.

`Negative(R, r)`

The index of the negative of the rth (co)root of the root system R. In other words, if $s =$ `Negative(R,`r`)` then $\alpha_s = -\alpha_r$.

Example H102E13

```
> R := RootSystem("G2");
> Sum(R, 1, Negative(R,5));
10
> IsPositive(R, 10);
false
> Negative(R, 10);
4
> P := PositiveRoots(R);
> P[1] - P[5] eq -P[4];
true
```

`RootHeight(R, r)`

`CorootHeight(R, r)`

The height of the rth (co)root of the root system R, i.e. the sum of the coefficients of α_r (respectively, $\alpha_r^\star$) with respect to the simple (co)roots.

`RootNorms(R)`

`CorootNorms(R)`

The sequence of squares of the lengths of the (co)roots of the root system R.

`RootNorm(R, r)`

`CorootNorm(R, r)`

The square of the length of the rth (co)root of the root system R.

`IsLongRoot(R, r)`

Returns `true` if, and only if, the rth root of the root system R is long. This only makes sense for irreducible crystallographic root systems. Note that for non-reduced root systems, the roots which are not indivisible are actually longer than the long ones.

`IsShortRoot(R, r)`

Returns `true` if, and only if, the rth root of the root system R is short. This only makes sense for irreducible crystallographic root systems.

`IsIndivisibleRoot(R, r)`

Returns `true` if, and only if, the rth root of the root system R is indivisible, ie, $\alpha_r/2$ is not a root.

`LeftString(R, r, s)`

Indices in the crystallographic root system R of the left string through α_s in the direction of α_r, i.e. the indices of $\alpha_s - \alpha_r, \alpha_s - 2\alpha_r, \ldots, \alpha_s - p\alpha_r$. In other words, this returns the sequence $[r_1, \ldots, r_p]$ where $\alpha_{r_i} = \alpha_s - i\alpha_r$ and $\alpha_s - (p+1)\alpha_r$ is not a root. We require that $\alpha_r \neq \pm\alpha_s$.

`RightString(R, r, s)`

Indices in the crystallographic root system R of the left string through α_s in the direction of α_r, i.e. the indices of $\alpha_s + \alpha_r, \alpha_s + 2\alpha_r, \ldots, \alpha_s + q\alpha_r$. In other words, this returns the sequence $[r_1, \ldots, r_q]$ where $\alpha_{r_i} = \alpha_s + i\alpha_r$ and $\alpha_s + (q+1)\alpha_r$ is not a root. We require that $\alpha_r \neq \pm\alpha_s$.

`LeftStringLength(R, r, s)`

The largest p such that $\alpha_s - p\alpha_r$ is a root. We require that the root system R be crystallographic and $\alpha_s \neq \pm\alpha_r$.

`RightStringLength(R, r, s)`

The largest q such that $\alpha_s + q\alpha_r$ is a root. We require that the root system R be crystallographic and $\alpha_s \neq \pm\alpha_r$.

Example H102E14

```
> R := RootSystem("G2");
> RootHeight(R, 5);
4
> F := CoxeterForm(R);
> v := Root(R, 5);
> (v*F, v) eq RootNorm(R, 5);
true
> IsLongRoot(R, 5);
true
> LeftString(R, 1, 5);
[ 4, 3, 2 ]
> roots := Roots(R);
> for i in [1..3] do
>   RootPosition(R, roots[5]-i*roots[1]);
> end for;
4
3
2
> R := RootSystem("BC2");
> Root(R,2), IsIndivisibleRoot(R,2);
(0 1) true
> Root(R,4), IsIndivisibleRoot(R,4);
(0 2) false
```

`AdditiveOrder(R)`

An additive order on the positive roots of the root system R, ie. a sequence containing the numbers $1, \ldots, N$ in some order so that $\alpha_r + \alpha_s = \alpha_t$ implies t is between r and s. This is computed using the techniques of [Pap94].

`IsAdditiveOrder(R, Q)`

Returns `true` if, and only if, the sequence Q gives an additive order on a set of positive roots of the root system R. Q must be a sequence of integers in the range $[1..N]$, where N is the number of positive roots of R, with no gaps or repeats.

Example H102E15

```
> R := RootSystem("A5");
> a := AdditiveOrder(R);
> Position(a, 2);
6
> Position(a, 3);
10
> Position(a, Sum(R, 2, 3));
7
```

102.6 Building Root Systems

`sub< R | a >`

The root subsystem of the root system R generated by the roots $\alpha_{a_1}, \ldots, \alpha_{a_k}$ where $a = \{a_1, \ldots, a_k\}$ is a set of integers.

`sub< R | s >`

The root subsystem of the root system R generated by the roots $\alpha_{s_1}, \ldots, \alpha_{s_k}$ where $s = [s_1, \ldots, s_k]$ is a *sequence* of integers. In this version the roots must be simple in the root subsystem (i.e. none of them may be a summand of another), otherwise an error is signalled. The simple roots will appear in the subsystem in the given order.

`R1 subset R2`

Returns `true` if and only if the root system R_1 is a subset of the root system R_2. If true, returns an injection as sequence of roots as second return value.

`R1 + R2`

`DirectSum(R1, R2)`

The direct sum of the root systems R_1 and R_2. The root space of the result is the direct sum of the root spaces of R_1 and R_2.

`R1 join R2`

The union of the root systems R_1 and R_2. The root systems must have the same root space, which will also be the root space of the result.

Example H102E16

```
> R  := RootSystem("A1A1");
> R1 := sub<R|[1]>;
> R2 := sub<R|[2]>;
> R1 + R2;
Root system of dimension 4 of type A1 A1
> R1 join R2;
Root system of dimension 2 of type A1 A1

> R1 := RootSystem("A3T2B4T3");
> R2 := RootSystem("T3G2T4BC3");
> R1 + R2;
Root system of dimension 24 of type A3 B4 G2 BC3
> R1 join R2;
Root system of dimension 12 of type A3 B4 G2 BC3
```

`DirectSumDecomposition(R)`

`IndecomposableSummands(R)`

The set of irreducible direct summands of the semisimple root system R.

`Dual(R)`

The dual of the root system R, obtained by swapping the roots and coroots.

`IndivisibleSubsystem(R)`

The root system consisting of all indivisible roots of the root system R.

Example H102E17

```
> R1 := RootSystem("H4");
> R2 := RootSystem("B4");
> R1 + Dual(R2);
Root system of type H4 C4
> R := RootSystem("BC2");
> I := IndivisibleSubsystem(R); I;
I: Root system of type B2
> I subset R;
true [ 1, 2, 3, 5, 7, 8, 9, 11 ]
```

102.7 Related Structures

In this section functions for creating other structures from a root system are briefly listed. The reader is referred to the appropriate chapters of the Handbook for more details.

`RootDatum(R)`

The (split) root datum corresponding to the root system R. The coefficients of the simple roots and coroots must be integral; otherwise an error is signalled. See Chapter 103

`CoxeterGroup(GrpFPCox, R)`

The Coxeter group with root system R. See Chapter 104. The braid group and pure braid group can be computed from the Coxeter group using the commands in Section 104.12.

`CoxeterGroup(R)`

`CoxeterGroup(GrpPermCox, R)`

The permutation Coxeter group with root system R. See Chapter 104.

`ReflectionGroup(R)`

`CoxeterGroup(GrpMat, W)`

The reflection group of the root system R. See Chapter 105.

`LieAlgebra(R, k)`

The Lie algebra of the root system R over the base ring k. See Chapter 106.

`MatrixLieAlgebra(R, k)`

The matrix Lie algebra of the root system R over the base ring k. See Chapter 106.

Example H102E18

```
> R := RootSystem("b3");
> SemisimpleType(LieAlgebra(R, Rationals()));
B3
> #CoxeterGroup(R);
48
```

102.8 Bibliography

[**Bou68**] N. Bourbaki. *Éléments de mathématique. Fasc. XXXIV. Groupes et algèbres de Lie. Chapitre IV: Groupes de Coxeter et systèmes de Tits. Chapitre V: Groupes engendrés par des réflexions. Chapitre VI: Systèmes de racines.* Hermann, Paris, 1968.

[**Pap94**] Paolo Papi. A characterization of a special ordering in a root system. *Proc. Amer. Math. Soc.*, 120(3):661–665, 1994.

103 ROOT DATA

103.1 Introduction 3089

103.1.1 Reflections 3089

103.1.2 Definition of a Split Root Datum 3090

103.1.3 Simple and Positive Roots . . . 3090

103.1.4 The Coxeter Group 3090

103.1.5 Nonreduced Root Data 3091

103.1.6 Isogeny of Split Reduced Root Data 3091

103.1.7 Extended Root Data 3092

103.2 Constructing Root Data . . 3092

RootDatum(N) 3094
RootDatum(C) 3096
RootDatum(D) 3096
RootDatum(A, B) 3096
IrreducibleRootDatum(X, n) 3097
StandardRootDatum(X, n) 3097
ToralRootDatum(n) 3098
TrivialRootDatum() 3098

103.2.1 Constructing Sparse Root Data . 3098

SparseRootDatum(N) 3098
SparseRootDatum(N) 3098
SparseRootDatum(C) 3098
SparseRootDatum(D) 3098
SparseRootDatum(R) 3098
SparseRootDatum(A, B) 3098
SparseIrreducibleRootDatum(X, n) 3098
SparseStandardRootDatum(X, n) 3098
SparseRootDatum(R) 3099
RootDatum(R) 3099

103.3 Operations on Root Data . . 3100

eq 3100
IsIsomorphic(R1, R2) 3100
IsCartanEquivalent(R1, R2) 3100
IsIsogenous(R1, R2) 3100
CartanName(R) 3101
TwistedCartanName(R) 3101
CoxeterDiagram(R) 3101
DynkinDiagram(R) 3101
CoxeterMatrix(R) 3101
CoxeterGraph(R) 3101
CartanMatrix(R) 3101
DynkinDigraph(R) 3101
GammaAction(R) 3102
GammaRootSpace(R) 3102
GammaCorootSpace(R) 3102
GammaOrbitOnRoots(R,r) 3103
GammaOrbitsOnRoots(R) 3103
PositiveGammaOrbitsOnRoots(R) 3103
NegativeGammaOrbitsOnRoots(R) 3103
ZeroGammaOrbitsOnRoots(R) 3103
GammaActionOnSimples(R) 3103
OrbitsOnSimples(R) 3103
DistinguishedOrbitsOnSimples(R) 3103
BaseRing(R) 3103
Rank(R) 3103
AbsoluteRank(R) 3103
RelativeRank(R) 3103
Dimension(R) 3103
TwistingDegree(R) 3103
AnisotropicSubdatum(R) 3103
CoxeterGroupOrder(R) 3105
GroupOfLieTypeOrder(R, q) 3105
GroupOfLieTypeFactoredOrder(R, q) 3105
FundamentalGroup(R) 3106
IsogenyGroup(R) 3106
CoisogenyGroup(R) 3106

103.4 Properties of Root Data . . 3107

IsFinite(R) 3107
IsIrreducible(R) 3107
IsAbsolutelyIrreducible(R) 3107
IsProjectivelyIrreducible(R) 3107
IsReduced(R) 3107
IsSemisimple(R) 3107
IsCrystallographic(R) 3107
IsSimplyLaced(R) 3108
IsAdjoint(R) 3108
IsWeaklyAdjoint(R) 3108
IsSimplyConnected(R) 3108
IsWeaklySimplyConnected(R) 3108
IsReduced(R) 3109
IsSplit(R) 3109
IsTwisted(R) 3109
IsQuasisplit(R) 3109
IsInner(R) 3109
IsOuter(R) 3109
IsAnisotropic(R) 3109

103.5 Roots, Coroots and Weights 3110

103.5.1 Accessing Roots and Coroots . . 3110

RootSpace(R) 3110
CorootSpace(R) 3110
FullRootLattice(R) 3110
FullCorootLattice(R) 3110
RootLattice(R) 3110
CorootLattice(R) 3110
IsRootSpace(V) 3111
IsCorootSpace(V) 3111
IsInRootSpace(v) 3111
IsCorootSpace(v) 3111
RootDatum(V) 3111
ZeroRootLattice(R) 3111
ZeroRootSpace(R) 3111

RelativeRootSpace(R) 3111
SimpleRoots(R) 3111
SimpleCoroots(R) 3111
NumberOfPositiveRoots(R) 3112
NumPosRoots(R) 3112
Roots(R) 3112
Coroots(R) 3112
PositiveRoots(R) 3112
PositiveCoroots(R) 3112
Root(R, r) 3112
Coroot(R, r) 3112
RootPosition(R, v) 3112
CorootPosition(R, v) 3112
BasisChange(R,v) 3112
IsInRootSpace(R,v) 3114
IsInCorootSpace(R,v) 3114
HighestRoot(R) 3114
HighestCoroot(R) 3114
HighestLongRoot(R) 3114
HighestLongCoroot(R) 3114
HighestShortRoot(R) 3114
HighestShortCoroot(R) 3114
RelativeRoots(R) 3114
PositiveRelativeRoots(R) 3114
NegativeRelativeRoots(R) 3114
SimpleRelativeRoots(R) 3114
RelativeRootDatum(R) 3115
GammaOrbitsRepresentatives(R, delta) 3115
CoxeterForm(R) 3117
DualCoxeterForm(R) 3117
103.5.2 Reflections 3117
SimpleReflectionMatrices(R) 3117
SimpleCoreflectionMatrices(R) 3117
ReflectionMatrices(R) 3117
CoreflectionMatrices(R) 3117
ReflectionMatrix(R, r) 3117
CoreflectionMatrix(R, r) 3117
SimpleReflectionPermutations(R) 3117
ReflectionPermutations(R) 3118
ReflectionPermutation(R, r) 3118
ReflectionWords(R) 3118
ReflectionWord(R, r) 3118
103.5.3 Operations and Properties for Root and Coroot Indices 3119
Sum(R, r, s) 3119
IsPositive(R, r) 3119
IsNegative(R, r) 3119
Negative(R, r) 3119
LeftString(R, r, s) 3119
RightString(R, r, s) 3119
LeftStringLength(R, r, s) 3119
RightStringLength(R, r, s) 3119
RootHeight(R, r) 3120
CorootHeight(R, r) 3120
RootNorms(R) 3120
CorootNorms(R) 3120
RootNorm(R, r) 3120
CorootNorm(R, r) 3120
IsLongRoot(R, r) 3120
IsShortRoot(R, r) 3120
IsIndivisibleRoot(R, r) 3120
RootClosure(R, S) 3121
AdditiveOrder(R) 3121
IsAdditiveOrder(R, Q) 3121
103.5.4 Weights 3122
WeightLattice(R) 3122
CoweightLattice(R) 3122
FundamentalWeights(R) 3122
FundamentalCoweights(R) 3122
IsDominant(R, v) 3123
DominantWeight(R, v) 3123
WeightOrbit(R, v) 3123
103.6 Building Root Data 3124
sub< > 3124
sub< > 3124
subset 3125
+ 3125
DirectSum(R1, R2) 3125
join 3125
DirectSumDecomposition(R) 3126
IndecomposableSummands(R) 3126
Dual(R) 3127
SimplyConnectedVersion(R) 3127
AdjointVersion(R) 3127
IndivisibleSubdatum(R) 3127
Radical(R) 3127
TwistedRootDatum(R) 3128
TwistedRootDatum(N) 3128
UntwistedRootDatum(R) 3129
SplitRootDatum(R) 3129
103.7 Morphisms of Root Data . . 3130
hom< > 3130
hom< > 3130
Morphism(R, S, phiX, phiY) 3130
Morphism(R, S, Q) 3130
DualMorphism(R, S, phiX, phiY) 3131
DualMorphism(R, S, Q) 3131
RootImages(phi) 3131
RootPermutation(phi) 3131
IsIsogeny(phi) 3131
IdentityMap(R) 3131
IdentityAutomorphism(R) 3131
103.8 Constants Associated with Root Data 3132
ExtraspecialPairs(R) 3132
NumExtraspecialPairs(R) 3132
ExtraspecialPair(R,r) 3132
ExtraspecialSigns(R) 3132
LieConstant_p(R, r, s) 3132
LieConstant_q(R, r, s) 3132
CartanInteger(R, r, s) 3132
LieConstant_N(R, r, s) 3132

LieConstant_epsilon(R, r, s) 3132
LieConstant_M(R, r, s, i) 3133
LieConstant_C(R, i, j, r, s) 3133
LieConstant_eta(R, r, s) 3133
StructureConstants(R) 3133

103.9 Related Structures 3134

RootSystem(R) 3134
CoxeterGroup(GrpFPCox, R) 3134
CoxeterGroup(R) 3134
CoxeterGroup(GrpPermCox, R) 3134
ReflectionGroup(R) 3134
CoxeterGroup(GrpMat, R) 3134
LieAlgebraHomorphism(phi,k) 3134
LieAlgebra(R, k) 3134
GroupOfLieType(R, k) 3134
GroupOfLieTypeHomomorphism(phi, k) 3134

103.10 Bibliography 3135

Chapter 103
ROOT DATA

103.1 Introduction

This chapter describes Magma functions for computing with (extended) root data. Root data are fundamental to Lie theory: Lie algebras (Chapter 106) and groups of Lie type (Chapter 109). Our description of split reduced root data follows [Dem70] and [Car93] except that reflections act on the right as in customary in MAGMA. Our description of extended root data follows [Sat71], [Sch69], and [Hal05]. Our description of split non-reduced root data follows [Bou68].

The closely related concept of a root system is discussed in Chapter 102. When working with Lie algebras or groups of Lie type, root data should be used. When working with Coxeter groups (Chapter 104) or reflection groups (Chapter 105), it is likely that only root systems are of interest.

103.1.1 Reflections

Let X and Y be free $\mathbf{Z}$-modules with bilinear pairing $\langle \circ, \circ \rangle : X \times Y \to \mathbf{Z}$ that identifies Y with the dual of X. Given nonzero $\alpha \in X$ and $\alpha^\star \in Y$, we define the $\mathbf{Z}$-linear map $s_\alpha : X \to X$ by

$$x s_\alpha = x - \langle x, \alpha^\star \rangle \alpha$$

and the $\mathbf{Z}$-linear map $s_\alpha^\star : Y \to Y$ by

$$y s_\alpha^\star = y - \langle \alpha, y \rangle \alpha^\star.$$

These maps are called *reflections* if one of the following equivalent properties hold: $\langle \alpha, \alpha^\star \rangle = 2$; ${s_\alpha}^2 = 1$; $\langle x s_\alpha, y s_\alpha^\star \rangle = \langle x, y \rangle$ for all $x \in X$ and $y \in Y$; $\alpha s_\alpha = -\alpha$. The map $s_\alpha^\star$ is also called a *coreflection*: this just means it is a reflection defined on Y instead of X. MAGMA functions for computing with reflections are described in Section 105.2.

If X has an inner product, then we can take $Y = X$ and use the inner product as our pairing. In MAGMA, X and Y are usually standard $\mathbf{Z}$-modules. However, it is sometimes useful to allow X and Y to be distinct sublattices of a standard lattice. The bilinear pairing is always given by the standard inner product: $\langle x, y \rangle = x y^T$.

103.1.2 Definition of a Split Root Datum

Suppose Φ is a finite subset of $X \setminus \{0\}$. For each α in Φ, suppose there is a corresponding $\alpha^\star$ in $Y \setminus \{0\}$; set $\Phi^\star = \{\alpha^\star \mid \alpha \in \Phi\}$. The datum $R = (X, \Phi, Y, \Phi^\star)$ is said to be a *(split) root datum* if the following conditions are satisfied for every α in Φ

1. s_α and $s_\alpha^\star$ are reflections;
2. Φ is closed under the action of s_α; and
3. $\Phi^\star$ is closed under the action of $s_\alpha^\star$.

The lattice X is called the *full root lattice* and Y the *full coroot lattice*. The vector space $X \otimes \mathbf{Q}$ is called the *root space* and $Y \otimes \mathbf{Q}$ the *coroot space*. The elements of Φ are called *roots* and the elements of $\Phi^\star$ are called *coroots*. A root datum is *reduced*, if $\alpha, \beta \in \Phi$ with β a scalar product of α implies $\alpha = \pm\beta$.

103.1.3 Simple and Positive Roots

A subset Δ of Φ is called a set of *simple roots* if

1. Δ is a basis for the rational span of the roots $\mathbf{Q}\Phi \leq \mathbf{Q} \otimes X$; and
2. $\Phi = \Phi^+ \cup \Phi^-$, where Φ^+ is the set of linear combinations of elements of Δ with nonnegative coefficients, and $\Phi^- = -\Phi^+$.

Every root datum has a set of simple roots. Simple roots are frequently called fundamental roots. The elements of Φ^+ are called *positive roots* and the elements of Φ^- *negative roots*. The coroots corresponding to the simple (resp. positive, negative) roots are the *simple* (respectively, *positive*, *negative*) *coroots*.

The *rank* of the root datum is the size of Δ, i.e. the dimension of the subspace $\mathbf{Q}\Phi$. The rank cannot be larger than the *dimension* of the root datum (i.e. the dimension of $\mathbf{Q} \otimes X$). If the rank and dimension are equal, the root datum is said to be *semisimple*.

Choose a basis $e_1, \ldots, e_d$ for X and a dual basis $f_1, \ldots, f_d$ for Y, so that $\langle e_i, f_j \rangle = \delta_{ij}$. A reduced root system is determined by a pair of integral matrices A and B where the rows of A are the simple roots and the rows of B are the corresponding coroots; i.e. $A_{ij} = \langle \alpha_i, f_j \rangle$ and $B_{ij} = \langle e_j, \alpha_i^\star \rangle$.

103.1.4 The Coxeter Group

The group W generated by the reflections s_α, for α a simple root, is a finite Coxeter group. The *Cartan matrix* of a root datum is

$$C = \left(\left\langle \, \alpha_i, \alpha_j^\star \, \right\rangle \right)_{i,j=1}^{n} = AB^t.$$

As in Chapter 101, the Cartan matrix is used to define the Coxeter matrix, Coxeter graph and Dynkin digraph of a root datum.

A *Coxeter form* is a W-invariant bilinear form on X. If R is reduced and irreducible, then the roots can have at most two different lengths with respect to this form. We call the roots *long* or *short* accordingly. The Coxeter form is normalised so that the short roots in each component have length one. Note that, even if $X = Y$, this form will generally not be the same as the pairing $\langle \circ, \circ \rangle$; however it can often be arranged for them to be the same (see `StandardRootSystem`).

103.1.5 Nonreduced Root Data

A root datum is *reduced*, if $\alpha, \beta \in \Phi$ with β a scalar product of α implies $\alpha = \pm\beta$. A root α with the property $2\alpha \notin \Phi$ is called *reduced*. A root α with the property $\frac{1}{2}\alpha \in \Phi$ is called *divisible*. If R is a root datum, then the set R_0 of indivisible roots in R form the *indivisible subsystem*.

Let R be a nonreduced irreducible root datum of rank n. It can be shown that R_0 is irreducible of type of type B_n and every root is either in R_0, or is two times a short root of R_0. The Cartan type of R in this case is BC_n.

Note that the Cartan matrix, Coxeter matrix, Coxeter diagram, Coxeter group and Dynkin diagram are the same for R and R_0. Thus, when creating a non-reduced root datum for a given Cartan matrix, Coxeter matrix, Coxeter diagram, Coxeter group or Dynkin diagram, one must specify the set of non-reduced fundamental roots. E.g., let C be a cartan matrix of type $B_2 \times B_3$. Then the set of nonreduced fundamental roots can be one of $\emptyset$, $\{2\}$, $\{5\}$ or $\{2, 5\}$, in which cases the root datum will be of types $B_2 \times B_3$, $BC_2 \times B_3$, $B_2 \times BC_3$ or $BC_2 \times BC_3$ respectively.

103.1.6 Isogeny of Split Reduced Root Data

The Dynkin digraph and dimension do not completely determine the isomorphism type of a split root datum, as the Coxeter graph and dimension do for a root system. Two root data with isomorphic Dynkin digraphs are said to be *Cartan equivalent*. We now describe the isomorphism classes within each Cartan equivalence class of split reduced irreducible root data. Since every semisimple reduced root datum is isogenous to a direct sum of irreducible root data, this immediately gives a classification of the split semisimple root data. Classifying nonsemisimple root data would be more complicated.

The *weights* of a root datum are the λ in $\mathbf{Q}\Phi \leq X \otimes \mathbf{Q}$ such that $\langle \lambda, \alpha^\star \rangle \in \mathbf{Z}$ for every coroot $\alpha^\star$. The weights form a lattice Λ called the *weight lattice*. We now have lattices $\mathbf{Z}\Phi \leq X \leq \Lambda$ (note that the second inclusion holds only for semisimple root data). The isomorphism class of a root datum in a fixed Cartan equivalence class is determined by the position of X between the root lattice $\mathbf{Z}\Phi$ and the weight lattice Λ. Alternatively, the isomorphism class is determined by the *isogeny group* $X/\mathbf{Z}\Phi$ within the *fundamental group* $\Lambda/\mathbf{Z}\Phi$. The fundamental group is determined by the Cartan matrix C: it is isometric to $\mathbf{Z}^n/\Theta$ where Θ is the lattice generated by the rows of C. The fundamental groups of the irreducible Cartan equivalence classes are

A_n: $\mathbf{Z}/(n+1)$;

B_n, C_n, E_7: $\mathbf{Z}/2$;

D_n: $\mathbf{Z}/4$ for n odd, $\mathbf{Z}/2 \times \mathbf{Z}/2$ for n even;

E_6: $\mathbf{Z}/3$;

E_8, F_4, G_2: trivial.

If $X = \mathbf{Z}\Phi$ the root datum is said to be *adjoint*; if $X = \Lambda$ it is said to be *simply connected*. The quotient $Y/\mathbf{Z}\Phi^\star$ is called the *coisogeny group*; in the semisimple case it is isomorphic to $\Lambda/\mathbf{Z}\Phi$.

103.1.7 Extended Root Data

An extended root datum is a split root datum $R = (X, \Phi, Y, \Phi^\star)$ and a permutation group Γ with actions on X and Y that respect the pairing $\langle \circ, \circ \rangle$.

Fix a set of simple roots Δ. Let $O(\chi)$ denote the orbit of $\chi \in X$ under the Γ-action. Then, for $\alpha \in \Phi$ either $O(\alpha)$ is contained in Φ^+, or it is contained in Φ^-, or the sum of the roots of $O(\alpha)$ is zero. We call $O(\alpha)$ a *positive*, *negative* or *zero orbit*, respectively. Put

$$X_0 := \{\chi \in X \mid \sum_{\gamma \in \Gamma} \chi^\gamma = 0\}.$$

Let $\Phi_0 := \Phi \cap X_0$ and $\Delta_0 := \Delta \cap X_0$. Then X_0 is a submodule of X, Φ_0 is a subsystem of Φ, and Δ_0 is a fundamental system of Φ_0. Note that Δ_0 is not necessarily a basis of X_0. Analogously, we define Y_0 and $\Phi_0^\star$. The subdatum $R_0 = (X_0, \Phi_0, Y_0, \Phi_0^\star)$ is called the *anisotropic subdatum* of R.

Set $\bar{X} := X/X_0$ and let $\pi : X \to \bar{X}$ be the standard projection. Then $\bar{X}$ is a free $\mathbf{Z}$-module and π is a homomorphism of modules. Let $\bar{\Phi}$ and $\bar{\Delta}$ be the images under π of $\Phi \setminus \Phi_0$ and $\Delta \setminus \Delta_0$, respectively. Then $\bar{\Phi}$ is a root system and $\bar{\Delta}$ is a fundamental system of it. We call $\bar{\Phi}$ the *relative root system* and $\bar{\Delta}$ the *relative fundamental system*. Note that $\bar{\Phi}$ need not be irreducible nor reduced even if Φ is. The rank of the relative system is $|\bar{\Delta}|$ and is called the *relative rank*, whereas the rank $|\Delta|$ of Φ is called the *absolute rank*. Let $\bar{\Phi}^+$ and $\bar{\Phi}^-$ denote the images under π of $\Phi^+ \setminus \Phi_0$ and $\Phi^- \setminus \Phi_0$. When $X_0 = X$, the relative root system is an empty set and the form is called *anisotropic*.

Each $\gamma \in \Gamma$ acts on X by $\chi \mapsto \chi^{\sigma w}$ for some unique $w \in W$ and σ a Dynkin diagram symmetry. By $\alpha \mapsto \alpha^\sigma$ for $\alpha \in \Delta$ we define the $[\Gamma]$-action on Δ. The extended root datum is called *inner* if the $[\Gamma]$-action is trivial and *outer* otherwise. The orbits of the $[\Gamma]$-action, that are not contained in X_0 are called *distinguished*.

An extended root datum is called *twisted* if the Γ-action is not trivial.

The (split) Cartan name of an extended root datum is the name of the corresponding split root datum. An extended root datum is *absolutely irreducible* if the corresponding split datum is irreducible. It is irreducible if there is no direct sum decomposition of the split datum which is preserved under the action of Γ. The *twisted Cartan name* of a root datum is the Cartan name, with extra information describing the twist. The name ${}^m X_{n,e}$ indicates a root datum with split Cartan name X_n, where the kernel of the $[\Gamma]$-action has index m in Γ, and e is the rank of the relative root system. The twisted Cartan name describes absolutely irreducible root data up to isomorphism. This is not true for simple root data however.

103.2 Constructing Root Data

We first describe some optional parameters that are common to many functions described below.

`Isogeny` ANY *Default :* "*Ad*"

The optional parameter `Isogeny` specifies the isomorphism class of the root datum within the Cartan equivalence class (see Subsection 103.1.6). For irreducible Cartan names, `Isogeny` can be one of the following:

1. A string: `"Ad"` for adjoint or `"SC"` for simply connected.
2. An integer giving the size of the isogeny subgroup within the fundamental group. The root datum must be absolutely irreducible. This does not work in type D_n with n even and `Isogeny` $= 2$, since in this case there are three distinct isomorphism classes (see the example below to create these data).
3. An injection of an abelian group into the fundamental group.

For compound Cartan names, `Isogeny` can be a string (`"Ad"` or `"SC"`); an injection into the fundamental group; or a list of strings, integers and injections (one for each direct summand).

`Signs` ANY *Default :* 1

Many of the constants associated with root data depend on the choice of the sign ϵ_{rs} for each extraspecial pair (r, s). This parameter allows the user to fix these signs for the root datum R by giving a sequence s of length `NumExtraspecialPairs(R)` consisting of integers 1 or -1. It is also possible to set `Signs` to 1 instead of a sequence of all 1 and to -1 instead of a sequence of all -1.

`Twist` ANY *Default :* 1

This optional parameter defines a Γ-action of an extended root datum and will accept the following values:

1. a homomorphism from Γ into `Sym(2*N)`, where N is the number of positive roots, specifying the action of Γ on the (co)roots. (Only for semisimple root data).
2. an integer i giving the order of Γ, e.g., $1, 2, 3, 6$ for ${}^1D_4$, ${}^2D_4$, ${}^3D_4$, ${}^6D_4$ (only if $i = 1$ or the root datum is irreducible).
3. $\langle D, i\rangle$, where D is a set of distinguished orbits as sets of integers and i (integer) is the order of the Dynkin diagram symmetry involved (only for irreducible root data).
4. $\langle \Gamma, ims\rangle$, where Γ is the acting group and ims define images either as permutations of the simple roots or as permutation of all roots (only for semisimple root data).
5. $\langle \Gamma, imsR, imsC\rangle$, where Γ is the acting group and $imsR$ ($imsC$) is a sequence of matrices defining the action of Γ on the root space (coroot space).

`Nonreduced` SETENUM *Default :* {}

The optional argument `Nonreduced` is used to give the set of indices of the nonreduced simple roots. Note that a root datum cannot be both twisted and nonreduced.

`RootDatum(N)`

`Isogeny`	ANY	*Default :* "*Ad*"
`Signs`	ANY	*Default :* 1
`Twist`	ANY	*Default :* 1

A root datum with Cartan name given by the string N (see Section 101.6). In addition to the possible Cartan names described in Section 101.6, this function will also accept `"Tn"` as a component of the Cartan name, which stands for an n-dimensional toral subdatum. Note, however, that this addition is for input only and will not appear in the string returned by `CartanName` when applied to the resulting root datum (see example below).

If the optional parameter `Isogeny` is a list, its length should be equal to the total number of components. Entries of this list corresponding to toral components will be ignored.

If the corresponding Coxeter group is infinite affine, an error is flagged.

Example H103E1

Examples of adjoint and simply connected irreducible root data.

```
> RootDatum("E6");
Adjoint root datum of type E6
> RootDatum("E6" : Isogeny := "SC");
Simply connected root datum of type E6
```

With nonirreducible root data the isogeny can be given as a list.

```
> R := RootDatum("A5 B3" : Isogeny := [* 3, "Ad" *]);
> R : Maximal;
Root datum of type A5 B3 with simple roots
[1 0 0 0 0 0 0 0]
[0 1 0 0 0 0 0 0]
[0 0 1 0 0 0 0 0]
[0 0 0 1 0 0 0 0]
[1 2 0 1 3 0 0 0]
[0 0 0 0 0 1 0 0]
[0 0 0 0 0 0 1 0]
[0 0 0 0 0 0 0 1]
and simple coroots
[ 2 -1  0  0  0  0  0  0]
[-1  2 -1  0 -1  0  0  0]
[ 0 -1  2 -1  1  0  0  0]
[ 0  0 -1  2 -1  0  0  0]
[ 0  0  0 -1  1  0  0  0]
[ 0  0  0  0  0  2 -1  0]
[ 0  0  0  0  0 -1  2 -1]
[ 0  0  0  0  0  0 -2  2]
>
```

```
> RootDatum("E6 A3 B4" : Isogeny := "SC");
Simply connected root datum of type E6 A3 B4
```

Nonsemisimple root data can be constructed by specifying a central torus.

```
> R := RootDatum("B3 T2 A2" : Isogeny := [* "SC", 0, "Ad" *]);
> R;
R: Root datum of type B3 A2
> Dimension(R), Rank(R);
7 5
> SimpleCoroots(R);
[ 1  0  0  0  0  0  0]
[ 0  1  0  0  0  0  0]
[ 0  0  1  0  0  0  0]
[ 0  0  0  0  0  2 -1]
[ 0  0  0  0  0 -1  2]
```

The following code creates the three root data of type D_6 with isogeny groups of size 2 using injections into the fundamental group.

```
> G< a, b > := FundamentalGroup("D6");
> G;
Abelian Group isomorphic to Z/2 + Z/2
Defined on 2 generators
Relations:
    2*a = 0
    2*b = 0
> _, inj1 := sub< G | a >;
> R1 := RootDatum("D6" : Isogeny := inj1);
> _, inj2 := sub< G | b >;
> R2 := RootDatum("D6" : Isogeny := inj2);
> _, inj3 := sub< G | a*b >;
> R3 := RootDatum("D6" : Isogeny := inj3);
```

Example H103E2

Examples of extended root data:

```
> R := RootDatum("A5" : Twist := 2 ); R;
R: Twisted adjoint root datum of type 2A5,3
> R eq RootDatum("A5" : Twist := < Sym(2), [Sym(5)|(1,5)(2,4)] > );
true
> R eq RootDatum("A5" : Twist := < {{1,5},{2,4},{3}}, 2 > );
true
> RootDatum("D4" : Twist := 1);
Adjoint root datum of type D4
> RootDatum("D4" : Twist := 2);
Twisted adjoint root datum of type 2D4,3
> RootDatum("D4" : Twist := 3);
Twisted adjoint root datum of type 3D4,2
```

```
> RootDatum("D4" : Twist := 6);
Twisted adjoint root datum of type 6D4,2
>
> R := RootDatum("A2");
> TwistedRootDatum(R : Twist := 2);
Twisted adjoint root datum of type 2A2,1
```

`RootDatum(C)`

`Isogeny`	ANY	*Default :* "*Ad*"
`Signs`	ANY	*Default :* 1
`Twist`	ANY	*Default :* 1
`Nonreduced`	SETENUM	*Default :* {}

A semisimple root datum with crystallographic Cartan matrix C. If the corresponding Coxeter group is infinite, an error is flagged.

`RootDatum(D)`

`Isogeny`	ANY	*Default :* "*Ad*"
`Signs`	ANY	*Default :* 1
`Twist`	ANY	*Default :* 1
`Nonreduced`	SETENUM	*Default :* {}

A semisimple root datum with Dynkin digraph D. If the corresponding Coxeter group is infinite, an error is flagged.

`RootDatum(A, B)`

`Signs`	ANY	*Default :* 1
`Twist`	ANY	*Default :* 1
`Nonreduced`	SETENUM	*Default :* {}

The root datum with simple roots given by the rows of the matrix A and simple coroots given by the rows of the matrix B. The matrices A and B must have the following properties:

1. A and B must be integral matrices with the same number of rows and the same number of columns;
2. the number of columns must be at least the number of rows; and
3. AB^t must be the Cartan matrix of a finite Coxeter group.

Example H103E3

An example of a nonsemisimple root system of type G_2:

```
> A := Matrix(2,3, [1,-1,0, -1,1,-1]);
> B := Matrix(2,3, [1,-1,1, 0,1,-1]);
> RootDatum(A, B);
Root datum of type G2
```

An example of a non-reduced root datum and usage of `Nonreduced` argument:

```
> C := CoxeterMatrix("B2B2");
> RootDatum(C);
Adjoint root datum of type B2 B2
> RootDatum(C : Nonreduced:={2});
Adjoint root datum of type BC2 B2
> RootDatum(C : Nonreduced:={4});
Adjoint root datum of type B2 BC2
> RootDatum(C : Nonreduced:={2,4});
Adjoint root datum of type BC2 BC2
```

`IrreducibleRootDatum(X, n)`

`Signs`	ANY	*Default :* 1
`Twist`	ANY	*Default :* 1

The irreducible root datum with Cartan name X_n.

`StandardRootDatum(X, n)`

`Signs`	ANY	*Default :* 1
`Twist`	ANY	*Default :* 1

The standard root datum with Cartan name X_n, i.e. the root datum with the standard inner product equal to the Coxeter form up to a constant. For technical reasons, this is only possible for the classical types, i.e. X must be `"A"`, `"B"`, `"C"`, or `"D"`. Note that the standard root datum is not semisimple for type A_n.

Example H103E4

These functions are useful in loops.

```
> for X in ["A","B","G"] do
>   print NumPosRoots(IrreducibleRootDatum(X, 2));
> end for;
3
4
6
```

`ToralRootDatum(n)`

`Twist` ANY *Default :* 1

The toral root datum of dimension n, i.e., the n-dimensional root datum with no roots or coroots.

Example H103E5

Toral root datum of dimension 3 and a twisted version of it:

```
> ToralRootDatum(3);
Toral root datum of dimension 3
> M := Matrix(Rationals(),3,3,[0,1,0,1,0,0,0,0,1]);M;
[0 1 0]
[1 0 0]
[0 0 1]
> ToralRootDatum(3 : Twist := <Sym(2),[M],[M]>);
Twisted toral root datum of dimension 3
```

`TrivialRootDatum()`

The trivial root datum of dimension 0.

103.2.1 Constructing Sparse Root Data

Sparse root data differ from the usual root data only in the internal representation of the objects. The internal representation is less memory expensive and requires less time for creation. Sparse root data have type `RootDtmSprs`, which is a subcategory of `RootDtm`.

There are some limitation on the root data which can have sparse representation. First, sparse representation only makes sense for classical root data, that is of types A, B, C and D. At the moment only root data with a connected Coxeter diagram may have sparse representation and no twisted sparse root data can be constructed. T

`SparseRootDatum(N)`

`SparseRootDatum(N)`

`SparseRootDatum(C)`

`SparseRootDatum(D)`

`SparseRootDatum(R)`

`SparseRootDatum(A, B)`

`SparseIrreducibleRootDatum(X, n)`

`SparseStandardRootDatum(X, n)`

These functions have the same syntax as their counterparts without the "Sparse" in the name (see Section 103.2). The root datum returned has sparse representation. See [CHM08] for the algorithms used to construct sparse root data.

Example H103E6

```
> SparseRootDatum("A2");
Sparse adjoint root datum of dimension 2 of type A2
> SparseStandardRootDatum("A", 2);
Sparse root datum of dimension 3 of type A2
> SparseRootDatum("A2") eq RootDatum("A2");
true
```

`SparseRootDatum(R)`

Return a sparse root datum equal to the root datum R.

`RootDatum(R)`

Return a non-sparse root datum equal to the root datum R.

Example H103E7

Due to the restrictions mentioned above, some operations that create new root data, will return a non-sparse root datum even though the input was sparse.

```
> R := SparseRootDatum("A2");
> T := ToralRootDatum(3);
> R+T;
Sparse root datum of dimension 5 of type A2
> R+R;
Adjoint root datum of dimension 4 of type A2 A2
```

103.3 Operations on Root Data

`R1 eq R2`

Returns `true` if, and only if, R_1 and R_2 are identical root data.

`IsIsomorphic(R1, R2)`

Returns `true` if, and only if, R_1 and R_2 are isomorphic root data. If `true`, the second value returned is a sequence giving the simple root of R_2 corresponding to each simple root of R_1, and the third value returned is an isomorphism $R_1 \to R_2$. This function is currently only implemented for semisimple root data.

`IsCartanEquivalent(R1, R2)`

Returns `true` if, and only if, the root data R_1 and R_2 are Cartan equivalent, i.e. they have isomorphic Dynkin diagrams. If `true`, the second value returned is a sequence giving the simple root of R_2 corresponding to each simple root of R_1.

`IsIsogenous(R1, R2)`

Returns `true` if, and only if, R_1 and R_2 are isogenous root data. If `true`, the subsequent values returned are: a sequence giving the simple root of R_2 corresponding to each simple root of R_1, the corresponding adjoint root datum R_{ad}, the morphisms $R_{ad} \to R_1$ and $R_{ad} \to R2$, the corresponding simply connected root datum R_{sc}, and the morphisms $R_1 \to R_{sc}$ and $R_2 \to R_{sc}$.

Example H103E8

An example of isogenous root data:

```
> R1 := RootDatum("A3");
> R2 := RootDatum("A3" : Isogeny := "SC");
> R1 eq R2;
false
> IsIsomorphic(R1, R2);
false
> IsCartanEquivalent(R1, R2);
true [ 1, 2, 3 ]
> IsIsogenous(R1, R2);
true [ 1, 2, 3 ]
Adjoint root datum of type A3
Mapping from: RootDtm: ad to RootDtm: ad
Mapping from: RootDtm: ad to RootDtm: sc
Simply connected root datum of type A3
Mapping from: RootDtm: ad to RootDtm: sc
Mapping from: RootDtm: sc to RootDtm: sc
```

An example of distinct isomorphic root data:

```
> C := CartanMatrix("B2");
> R1 := RootDatum(C);
```

```
> R2 := RootDatum(Transpose(C));
> R1; R2;
Adjoint root datum of type B2
Adjoint root datum of type C2
> R1 eq R2;
false
> IsIsomorphic(R1, R2);
true [ 2, 1 ]
```

`CartanName(R)`

The Cartan name of the root datum R (Section 101.6).

`TwistedCartanName(R)`

The twisted Cartan name of the root datum R. E.g., `"2A_3,2"`.

`CoxeterDiagram(R)`

Print the Coxeter diagram of the root datum R (Section 101.6).

`DynkinDiagram(R)`

Print the Dynkin diagram of the root datum R (Section 101.6).

`CoxeterMatrix(R)`

The Coxeter matrix of the root datum R (Section 101.2).

`CoxeterGraph(R)`

The Coxeter graph of the root datum R (Section 101.3).

`CartanMatrix(R)`

The Cartan matrix of the root datum R (Section 101.4).

`DynkinDigraph(R)`

The Dynkin digraph of the root datum R (Section 101.5).

Example H103E9

```
> R := RootDatum("F4");
> DynkinDiagram(R);
F4    1 - 2 =>= 3 - 4
> CoxeterDiagram(R);
F4    1 - 2 === 3 - 4
```

Example H103E10

```
> R := RootDatum("G2");
> RootSpace(R);
Standard Lattice of rank 2 and degree 2
> CorootSpace(R);
Standard Lattice of rank 2 and degree 2
> // Add RootLattice, CorootLattice.
> // and maybe move (Co)RootSpace and (Co)RootLattice
> // to after introducing them
> SimpleRoots(R);
[1 0]
[0 1]
> SimpleCoroots(R);
[ 2 -3]
[-1  2]
> CartanMatrix(R);
[ 2 -1]
[-3  2]
> Rank(R) eq Dimension(R);
true
```

`GammaAction(R)`

The Γ-action of the root datum R. This is a record consisting of four elements: `gamma` is the Group Γ acting on R, `perm_ac` is the homomorphism defining the permutation action of Γ on the set of all roots of R, finally `mats_rt` and `mats_co` are sequences of matrices defining the action of Γ on the root and coroot spaces of R.

`GammaRootSpace(R)`

`GammaCorootSpace(R)`

Given a root datum R, create the fixed space of Γ acting on the (co)root space $V = \mathbf{Q} \otimes X$ of R as well as the embedding in V.

`GammaOrbitOnRoots(R,r)`

The orbit through the rth root of the Γ-action on the root datum R.

`GammaOrbitsOnRoots(R)`
`PositiveGammaOrbitsOnRoots(R)`
`NegativeGammaOrbitsOnRoots(R)`
`ZeroGammaOrbitsOnRoots(R)`

The sequence of all (respectively positive, negative and zero) orbits of the Γ-action on the root datum R (Section 103.1.7).

`GammaActionOnSimples(R)`

The $[\Gamma]$-action on the simple (co)roots of the root datum R. (Section 103.1.7). This function was called `GammaActionPi` in the last release.

`OrbitsOnSimples(R)`

The sequence of all orbits of the $[\Gamma]$-action on the simple (co)roots of the root datum R (Section 103.1.7). This function was called `OrbitsPi` in the last release.

`DistinguishedOrbitsOnSimples(R)`

The sequence of distinguished orbits of the $[\Gamma]$-action on the simple (co)roots of the root datum R (Section 103.1.7). This function was called `DistinguishedOrbitsPi` in the last release.

`BaseRing(R)`

The base ring of the root datum R is the field of rational numbers.

`Rank(R)`
`AbsoluteRank(R)`

The (absolute) rank of the root datum R, i.e. the number of simple (co)roots.

`RelativeRank(R)`

The relative rank of the root datum R, i.e. the number of simple (co)roots of the relative root system. This is the same as absolute rank for split root data.

`Dimension(R)`

The dimension of the root datum R, i.e. the dimension of the (co)root space. This is at least as large as the rank, with equality when R is semisimple.

`TwistingDegree(R)`

The twisting degree of the root datum R, i.e. the order of Γ divided by the kernel of the $[\Gamma]$-action.

`AnisotropicSubdatum(R)`

The anisitropic subdatum of the root datum R.

Example H103E11

Consider the twisted root datum of type ${}^2A_{3,1}$ with distinguished orbit $\{2\}$:

```
> R := RootDatum( "A3" : Twist := < {{2}}, 2 > );
```

First, print out the action of Γ on the root datum:

```
> GammaAction(R);
rec<recformat<gamma: GrpPerm, perm_ac: HomGrp, mats_rt, mats_co> |
     gamma := Permutation group acting on a set of cardinality 4
     Order = 4 = 2^2
          (1, 2, 3, 4),
     perm_ac := Homomorphism of GrpPerm: $, Degree 4, Order 2^2 into GrpPerm: $,
     Degree 12, Order 2^10 * 3^5 * 5^2 * 7 * 11 induced by
          (1, 2, 3, 4) |--> (1, 3, 7, 9)(2, 4, 6, 5)(8, 10, 12, 11),
     mats_rt := [
          [ 0  0  1]
          [ 1  1  0]
          [-1  0  0]
     ],
     mats_co := [
          [ 0  0  1]
          [ 0  1  0]
          [-1  1  0]
     ]
     >
```

Compute the orbits of the Γ-action:

```
> PositiveGammaOrbitsOnRoots(R);
[
     GSet{ 2, 4, 5, 6 }
]
> NegativeGammaOrbitsOnRoots(R);
[
     GSet{ 8, 10, 11, 12 }
]
> ZeroGammaOrbitsOnRoots(R);
[
     GSet{ 1, 3, 7, 9 }
]
> &+[ Root(R,r) : r in ZeroGammaOrbitsOnRoots(R)[1] ];
(0 0 0)
```

Compute the $[\Gamma]$-action and its orbits:

```
> GammaActionOnSimples(R);
Homomorphism of GrpPerm: $, Degree 4, Order 2^2 into GrpPerm: $,
Degree 3, Order 2 * 3 induced by
     (1, 2, 3, 4) |--> (1, 3)
```

```
> OrbitsOnSimples(R);
[
     GSet{ 2 },
     GSet{ 1, 3 }
]
> DistinguishedOrbitsOnSimples(R);
[
     GSet{ 2 }
]
```

Absolute and relative rank and the twisting degree, as well as their appearance in the name of the root datum:

```
> AbsoluteRank(R);
3
> RelativeRank(R);
1
> TwistingDegree(R);
2
> R;
R: Twisted adjoint root datum of type 2A3,1
```

anisotropic subdatum:

```
> A := AnisotropicSubdatum(R); A;
A: Twisted root datum of type 2(A1 A1)2,0
> GammaAction(A)`perm_ac;
Homomorphism of GrpPerm: $, Degree 4, Order 2^2 into GrpPerm: $,
Degree 4, Order 2^2 induced by
     (1, 2, 3, 4) |--> (1, 2, 3, 4)
```

`CoxeterGroupOrder(R)`

The order of the (split) Coxeter group of the root datum R.

`GroupOfLieTypeOrder(R, q)`

The order of the group of Lie type with split root datum R over the field of cardinality q.

`GroupOfLieTypeFactoredOrder(R, q)`

The factored order of the group of Lie type with split root datum R over the field of order q.

Example H103E12

As well as accepting a specific prime power, these functions also take an indeterminate so that the generic order formula can be computed.

```
> P<q> := PolynomialRing(Integers());
> R := RootDatum("F4");
> GroupOfLieTypeFactoredOrder(R, q);
[
    <q - 1, 4>,
    <q, 24>,
    <q + 1, 4>,
    <q^2 - q + 1, 2>,
    <q^2 + 1, 2>,
    <q^2 + q + 1, 2>,
    <q^4 - q^2 + 1, 1>,
    <q^4 + 1, 1>
]
>
> R := RootDatum("B2");
> ord := GroupOfLieTypeOrder(R, q);
> forall{ q : q in [2..200] | not IsPrimePower(q) or
>   Evaluate(ord, q) eq GroupOfLieTypeOrder(R, q) };
true
```

`FundamentalGroup(R)`

The fundamental group $\Lambda/\mathbf{Z}\Phi$ of the root datum R together with the projection $\Lambda \to \Lambda/\mathbf{Z}\Phi$. See Subsection 103.1.6.

`IsogenyGroup(R)`

The isogeny group $X/\mathbf{Z}\Phi$ of the root datum R together with the projection $X \to X/\mathbf{Z}\Phi$. If R is semisimple, the injection $X/\mathbf{Z}\Phi \to \Lambda/\mathbf{Z}\Phi$ is also returned. See Subsection 103.1.6.

`CoisogenyGroup(R)`

The coisogeny group $Y/\mathbf{Z}\Phi^\star$ of the root datum R together with the projection $Y \to Y/\mathbf{Z}\Phi^\star$. If R is semisimple, the projection $Y/\mathbf{Z}\Phi^\star \to \Lambda/\mathbf{Z}\Phi$ is also returned. See Subsection 103.1.6.

Example H103E13

In the semisimple case, the fundamental group contains the isogeny group, with quotient isomorphic to the coisogeny group.

```
> R := RootDatum("A5" : Isogeny := 3);
> F := FundamentalGroup(R);
> G := IsogenyGroup(R);
> H := CoisogenyGroup(R);
> #G * #H eq #F;
true
```

Nonsemisimple root data have infinite isogeny groups.

```
> R := StandardRootDatum("A", 5);
> IsogenyGroup(R);
Abelian Group isomorphic to Z
Defined on 1 generator (free)
```

103.4 Properties of Root Data

`IsFinite(R)`

Returns `true` for any root datum R.

`IsIrreducible(R)`

Returns `true` if, and only if, the root datum R is irreducible.

`IsAbsolutelyIrreducible(R)`

Returns `true` if, and only if, the split version of the root datum R is irreducible.

`IsProjectivelyIrreducible(R)`

Returns `true` if, and only if, the quotient of the root datum R modulo its radical is irreducible. This is equivalent for R to have a connected Coxeter diagram.

`IsReduced(R)`

Returns `true` if, and only if, the root datum R is reduced.

`IsSemisimple(R)`

Returns `true` if, and only if, the root datum R is semisimple, i.e. its rank is equal to its dimension.

`IsCrystallographic(R)`

Returns `true` for any root datum R.

`IsSimplyLaced(R)`

Returns `true` if, and only if, the root datum R is simply laced, i.e. its Dynkin diagram contains no multiple bonds.

`IsAdjoint(R)`

Returns `true` if, and only if, the root datum R is adjoint, i.e. its isogeny group is trivial.

`IsWeaklyAdjoint(R)`

Returns `true` if, and only if, the root datum R is weakly adjoint, i.e. its isogeny group is isomorphic to $\mathbf{Z}^n$, where n is $\dim(R) - \mathrm{rk}(R)$. Note that if R is semisimple then this function is identical to IsAdjoint.

`IsSimplyConnected(R)`

Returns `true` if, and only if, the root datum R is simply connected, i.e. its isogeny group is equal to the fundamental group, i.e. its coisogeny group is trivial.

`IsWeaklySimplyConnected(R)`

Returns `true` if, and only if, the root datum R is weakly simply connected, i.e. its coisogeny group is isomorphic to $\mathbf{Z}^n$, where n is $\dim(R) - \mathrm{rk}(R)$. Note that if R is semisimple then this function is identical to `IsSimplyConnected`.

Example H103E14

```
> R := RootDatum("A5 B2" : Isogeny := "SC");
> IsIrreducible(R);
false
> IsSimplyLaced(R);
false
> IsSemisimple(R);
true
> IsAdjoint(R);
false
```

For some of the exceptional isogeny classes, there is only one isomorphism class of root data, which is both adjoint and simply connected.

```
> R := RootDatum("G2");
> IsAdjoint(R);
true
> IsSimplyConnected(R);
true
```

There exist root data that are neither adjoint nor simply connected.

```
> R := RootDatum("A3" : Isogeny := 2);
> IsAdjoint(R), IsSimplyConnected(R);
```

```
false false
```

Finally, we demonstrate a case where the root datum is not adjoint, but is weakly adjoint.

```
> R := RootDatum("A2T1");
> IsAdjoint(R), IsWeaklyAdjoint(R);
false true
> Dimension(R), Rank(R);
3 2
> G := IsogenyGroup(R); G;
Abelian Group isomorphic to Z
Defined on 1 generator (free)
```

`IsReduced(R)`

Returns `true` if, and only if, the root datum R is reduced.

`IsSplit(R)`

Returns `true` if, and only if, the root datum R is split, i.e. the Γ-action is trivial.

`IsTwisted(R)`

Returns `true` if, and only if, the root datum R is twisted, i.e. the Γ-action is not trivial.

`IsQuasisplit(R)`

Returns `true` if, and only if, the root datum R is quasisplit, i.e. the anisotropic subdatum is trivial.

`IsInner(R)`
`IsOuter(R)`

Returns `true` if, and only if, the root datum R is inner (resp. outer).

`IsAnisotropic(R)`

Returns `true` if, and only if, the root datum R is anisotropic, i.e. when $X = X_0$.

103.5 Roots, Coroots and Weights

The roots are stored as an indexed set

$$\{@\ \alpha_1, \ldots, \alpha_N, \alpha_{N+1}, \ldots, \alpha_{2N}\ @\},$$

where $\alpha_1, \ldots, \alpha_N$ are the positive roots in an order compatible with height; and $\alpha_{N+1}, \ldots, \alpha_{2N}$ are the corresponding negative roots (i.e. $\alpha_{i+N} = -\alpha_i$). The simple roots are $\alpha_1, \ldots, \alpha_n$ where n is the rank.

Many of these functions have an optional argument `Basis` which may take one of the following values

1. `"Standard"`: the standard basis for the (co)root space. This is the default.
2. `"Root"`: the basis of simple (co)roots.
3. `"Weight"`: the basis of fundamental (co)weights (see Subsection 105.8.3 below).

103.5.1 Accessing Roots and Coroots

`RootSpace(R)`

`CorootSpace(R)`

The vector space containing the (co)roots of the root datum R, i.e. $X \otimes \mathbf{Q}$ (respectively, $Y \otimes \mathbf{Q}$).

`FullRootLattice(R)`

`FullCorootLattice(R)`

The lattice containing the (co)roots of the root datum R, i.e. X (respectively, Y). An inclusion map into the (co)root space of R is returned as the second value.

`RootLattice(R)`

`CorootLattice(R)`

The lattice spanned by the (co)roots of the root datum R. An inclusion map into the (co)root space of R is returned as the second value.

Example H103E15

The root space, full root lattice and the root lattice of the standard root datum of type A_2:

```
> R := StandardRootDatum("A",2);
> V := RootSpace(R);
> FullRootLattice(R);
Standard Lattice of rank 3 and degree 3
Mapping from: Standard Lattice of rank 3 and degree 3 to ModTupFld: V
> RootLattice(R);
Lattice of rank 2 and degree 3
Basis:
( 1 -1  0)
( 0  1 -1)
Mapping from: Lattice of rank 2 and degree 3 to ModTupFld: V
```

`IsRootSpace(V)`

`IsCorootSpace(V)`

Return `true` if, and only if, V is the (co)root space of some root datum.

`IsInRootSpace(v)`

`IsCorootSpace(v)`

Return `true` if, and only if, V is an element of the (co)root space of some root datum.

`RootDatum(V)`

If V is the (co)root space of some root datum, this returns the datum.

Example H103E16

```
> R := RootDatum("a3");
> V := RootSpace(R);
> v := V.1;
> IsRootSpace(V);
true
> RootDatum(V);
R: Adjoint root datum of dimension 3 of type A3
> IsInRootSpace(v);
true
```

`ZeroRootLattice(R)`

`ZeroRootSpace(R)`

For the given root datum R, return the lattice X_0 and the vector space $X_0 \otimes \mathbf{Q}$, respectively (see Section 103.1.7).

`RelativeRootSpace(R)`

For the given root datum R, return the vector space $\bar{X} = (X \otimes \mathbf{Q})/(X_0 \otimes \mathbf{Q})$ containing the relative roots (see Section 103.1.7). The projection from $X \otimes \mathbf{Q}$ onto $\bar{X}$ is returned as second return value.

`SimpleRoots(R)`

`SimpleCoroots(R)`

The simple (co)roots of the root datum R as the rows of a matrix, i.e. A (respectively, B).

`NumberOfPositiveRoots(R)`

`NumPosRoots(R)`

The number of positive roots of the root datum R. This is also the number of positive coroots. The total number of (co)roots is twice the number of positive (co)roots.

`Roots(R)`

`Coroots(R)`

`Basis` MonStgElt *Default :* "*Standard*"

The indexed set of (co)roots of the root datum R, i.e. $\{@\, \alpha_1, \ldots \alpha_{2N}\, @\}$ (respectively, $\{@\, \alpha_1^\star, \ldots \alpha_{2N}^\star\, @\}$).

`PositiveRoots(R)`

`PositiveCoroots(R)`

`Basis` MonStgElt *Default :* "*Standard*"

The indexed set of positive (co)roots of the root datum R, i.e. $\{@\, \alpha_1, \ldots \alpha_N\, @\}$ (respectively, $\{@\, \alpha_1^\star, \ldots \alpha_N^\star\, @\}$).

`Root(R, r)`

`Coroot(R, r)`

`Basis` MonStgElt *Default :* "*Standard*"

The rth (co)root α_r (respectively, $\alpha_r^\star$) of the root datum R.

`RootPosition(R, v)`

`CorootPosition(R, v)`

`Basis` MonStgElt *Default :* "*Standard*"

If v is a (co)root in the root datum R, return its index; otherwise return 0. These functions will try to coerce v into the appropriate lattice; v should be written with respect to the basis specified by the parameter `Basis`.

`BasisChange(R,v)`

`InBasis` MonStgElt *Default :* "*Standard*"

`OutBasis` MonStgElt *Default :* "*Standard*"

`Coroots` BoolElt *Default :* `false`

Changes the basis of the vector v contained in the space spanned by the (co)roots of the root datum R. The vector v is considered as an element of the root space by default. If the parameter `Coroots` is set to `true`, v is considered as an element of the coroot space. The optional arguments `InBasis` and `OutBasis` may take the same values as the parameter `Basis` as described at the beginning of the current section.

Example H103E17

```
> R := RootDatum("A3" : Isogeny := 2);
> Roots(R);
{@
    (1 0 0),
    (0 1 0),
    (1 0 2),
    (1 1 0),
    (1 1 2),
    (2 1 2),
    (-1  0  0),
    (0 -1  0),
    (-1  0 -2),
    (-1 -1  0),
    (-1 -1 -2),
    (-2 -1 -2)
@}
> PositiveCoroots(R);
{@
    (2 -1 -1),
    (-1  2  0),
    (0 -1  1),
    (1  1 -1),
    (-1  1  1),
    (1 0 0)
@}
> #Roots(R) eq 2*NumPosRoots(R);
true
> Coroot(R, 4);
(1  1 -1)
> Coroot(R, 4 : Basis := "Root");
(1 1 0)
> CorootPosition(R, [1,1,-1]);
4
> CorootPosition(R, [1,1,0] : Basis := "Root");
4
> BasisChange(R, [1,0,0] : InBasis:="Root");
(1 0 0)
> BasisChange(R, [1,0,0] : InBasis:="Root", Coroots);
( 2 -1 -1)
```

`IsInRootSpace(R,v)`

`IsInCorootSpace(R,v)`

Returns `true` if and only if the vector v is contained in the (co)root space of the root datum R.

`HighestRoot(R)`

`HighestCoroot(R)`

`Basis` MonStgElt *Default :* "*Standard*"

The unique (co)root of greatest height in the irreducible root datum R.

`HighestLongRoot(R)`

`HighestLongCoroot(R)`

`Basis` MonStgElt *Default :* "*Standard*"

The unique long (co)root of greatest height in the irreducible root datum R.

`HighestShortRoot(R)`

`HighestShortCoroot(R)`

`Basis` MonStgElt *Default :* "*Standard*"

The unique short (co)root of greatest height in the irreducible root datum R.

Example H103E18

```
> R := RootDatum("G2");
> HighestRoot(R);
(3 2)
> HighestLongRoot(R);
(3 2)
> HighestShortRoot(R);
(2 1)
```

`RelativeRoots(R)`

`PositiveRelativeRoots(R)`

`NegativeRelativeRoots(R)`

`SimpleRelativeRoots(R)`

The indexed set of all (resp. positive, negative, simple) relative roots of the root datum R. Note that the relative roots are returned in the order induced by the standard ordering on the (nonrelative) roots of R.

`RelativeRootDatum(R)`

The relative root datum of the root datum R.

`GammaOrbitsRepresentatives(R, delta)`

The preimage of a relative root δ is a disjoint union of Γ-orbits on the set of all roots of the root datum R. This intrinsic returns a sequence of representatives of these orbits.

Example H103E19

We first consider the twisted root datum of type ${}^2E_6$, which is quasisplit:

```
> DynkinDiagram(RootDatum("E6"));
E6    1 - 3 - 4 - 5 - 6
              |
              2
>
> R := RootDatum("E6" : Twist:=2 ); R;
R: Twisted adjoint root datum of type 2E6,4
> OrbitsOnSimples(R);
[
     GSet{ 2 },
     GSet{ 4 },
     GSet{ 1, 6 },
     GSet{ 3, 5 }
]
> DistinguishedOrbitsOnSimples(R) eq OrbitsOnSimples(R);
true
> AnisotropicSubdatum(R);
Twisted toral root datum of dimension 6
[]
> RR := RelativeRootDatum(R);RR;
RR: Adjoint root datum of type F4
> _,pi := RelativeRootSpace(R);
> DynkinDiagram(RR);
F4    1 - 2 =>= 4 - 3
> [ Position(Roots(RR), pi(Root(R,i)) ) : i in [1,6, 3,5, 4, 2]];
[ 3, 3, 4, 4, 2, 1 ]
```

now one with distinguished orbits $\{2\}$ and $\{4\}$:

```
> R := RootDatum("E6" : Twist := <{{2},{4}},2> ); R;
R: Twisted adjoint root datum of type 2E6,2
> OrbitsOnSimples(R);
[
     GSet{ 2 },
     GSet{ 4 },
     GSet{ 1, 6 },
     GSet{ 3, 5 }
```

```
]
> DistinguishedOrbitsOnSimples(R);
[
     GSet{ 2 },
     GSet{ 4 }
]
> AnisotropicSubdatum(R);
Twisted root datum of type 2(A2 A2)4,0
[ 1, 3, 5, 6, 7, 11, 37, 39, 41, 42, 43, 47 ]
> RR := RelativeRootDatum(R);RR;
RR: Adjoint root datum of type G2
> DynkinDiagram(RR);
G2    2 =<= 1
        3
> _,pi := RelativeRootSpace(R);
> [ Position(Roots(RR), pi(Root(R,i)) ) : i in [2,4]];
[ 1, 2 ]
```

and now the one with distinguished orbits $\{2\}$ and $\{1,6\}$, which has a non-reduced relative root datum:

```
> R := RootDatum("E6" : Twist := <{{2},{1,6}},2> ); R;
R: Twisted adjoint root datum of dimension 6 of type 2E6,2
> OrbitsOnSimples(R);
[
     GSet{ 2 },
     GSet{ 4 },
     GSet{ 1, 6 },
     GSet{ 3, 5 }
]
> DistinguishedOrbitsOnSimples(R);
[
     GSet{ 2 },
     GSet{ 1, 6 }
]
> AnisotropicSubdatum(R);
Twisted root datum of type 2A3,0
[ 3, 4, 5, 9, 10, 15, 39, 40, 41, 45, 46, 51 ]
> RR := RelativeRootDatum(R);RR;
RR: Adjoint root datum of type BC2
> DynkinDiagram(RR);
BC2    1 =>= 2
> _,pi := RelativeRootSpace(R);
> [ Position(Roots(RR), pi(Root(R,i)) ) : i in [2, 1,6]];
[ 1, 2, 2 ]
```

Finally, the twisted root Datum of type ${}^6D_{4,1}$:

```
> T := RootDatum( "D4" : Twist:=<{{2}},6> );
> T;
```

```
T: Twisted adjoint root datum of dimension 4 of type 6D4,1
> RelativeRootDatum(T);
Adjoint root datum of type BC1
> GammaOrbitsRepresentatives(T,1);
[ 11, 5 ]
> GammaOrbitsRepresentatives(T,2);
[ 12 ]
```

`CoxeterForm(R)`

`DualCoxeterForm(R)`

`Basis` MonStgElt *Default :* "*Standard*"

The matrix of an inner product on the (co)root space of the root datum R which is invariant under the action of the (co)roots. The inner product is normalised so that the short roots in each irreducible component have length one.

103.5.2 Reflections

The root α acts on the root space via the reflection s_α; the coroot $\alpha^\star$ acts on the coroot space via the coreflection $s^\star_\alpha$.

`SimpleReflectionMatrices(R)`

`SimpleCoreflectionMatrices(R)`

`Basis` MonStgElt *Default :* "*Standard*"

The sequence of matrices giving the action of the simple (co)roots of the root datum R on the (co)root space, i.e. the matrices of $s_{\alpha_1}, \ldots, s_{\alpha_n}$ (respectively, $s^\star_{\alpha_1}, \ldots, s^\star_{\alpha_n}$).

`ReflectionMatrices(R)`

`CoreflectionMatrices(R)`

`Basis` MonStgElt *Default :* "*Standard*"

The sequence of matrices giving the action of the (co)roots of the root datum R on the (co)root space, i.e. the matrices of $s_{\alpha_1}, \ldots, s_{\alpha_{2N}}$ (respectively, $s^\star_{\alpha_1}, \ldots, s^\star_{\alpha_{2N}}$).

`ReflectionMatrix(R, r)`

`CoreflectionMatrix(R, r)`

`Basis` MonStgElt *Default :* "*Standard*"

The matrix giving the action of the rth (co)root of the root datum R on the (co)root space, i.e. the matrix of s_{α_r} (respectively, $s^\star_{\alpha_r}$).

`SimpleReflectionPermutations(R)`

The sequence of permutations giving the action of the simple (co)roots of the root datum R on the (co)roots. This action is the same for roots and coroots.

`ReflectionPermutations(R)`

The sequence of permutations giving the action of the (co)roots of the root datum R on the (co)roots. This action is the same for roots and coroots.

`ReflectionPermutation(R, r)`

The permutation giving the action of the rth (co)root of the root datum R on the (co)roots. This action is the same for roots and coroots.

`ReflectionWords(R)`

The sequence of words in the simple reflections for all the reflections of the root datum R. These words are given as sequences of integers. In other words, if $a = [a_1, \ldots, a_l] =$ `ReflectionWords(R)[r]`, then $s_{\alpha_r} = s_{\alpha_{a_1}} \cdots s_{\alpha_{a_l}}$.

`ReflectionWord(R, r)`

The word in the simple reflections for the rth reflection of the root datum R. The word is given as a sequence of integers. In other words, if $a = [a_1, \ldots, a_l] =$ `ReflectionWord(R,r)`, then $s_{\alpha_r} = s_{\alpha_{a_1}} \cdots s_{\alpha_{a_l}}$.

Example H103E20

```
> R := RootDatum("A3" : Isogeny := 2);
> mx := ReflectionMatrix(R, 4);
> perm := ReflectionPermutation(R, 4);
> wd := ReflectionWord(R, 4);
> RootPosition(R, Root(R,2) * mx) eq 2^perm;
true
> perm eq &*[ ReflectionPermutation(R, r) : r in wd ];
true
>
> mx := CoreflectionMatrix(R, 4);
> CorootPosition(R, Coroot(R,2) * mx) eq 2^perm;
true
```

103.5.3 Operations and Properties for Root and Coroot Indices

`Sum(R, r, s)`

The index of the sum of the rth and sth roots in the root datum R, or 0 if the sum is not a root. In other words, if $t =$ `Sum(R,`r`,`s`)` $\neq 0$ then $\alpha_t = \alpha_r + \alpha_s$. The condition $\alpha_r \neq \pm\alpha_s$ must be satisfied.

`IsPositive(R, r)`

Returns `true` if, and only if, the rth (co)root of the root datum R is a positive root.

`IsNegative(R, r)`

Returns `true` if, and only if, the rth (co)root of the root datum R is a negative root.

`Negative(R, r)`

The index of the negative of the rth (co)root of the root datum R. In other words, if $s =$ `Negative(R,`r`)` then $\alpha_s = -\alpha_r$.

`LeftString(R, r, s)`

Indices in the root datum R of the left string through α_s in the direction of α_r, i.e. the indices of $\alpha_s - \alpha_r, \alpha_s - 2\alpha_r, \ldots, \alpha_s - p\alpha_r$. In other words, this returns the sequence $[r_1, \ldots, r_p]$ where $\alpha_{r_i} = \alpha_s - i\alpha_r$ and $\alpha_s - (p+1)\alpha_r$ is not a root. The condition $\alpha_r \neq \pm\alpha_s$ must be satisfied.

`RightString(R, r, s)`

Indices in the root datum R of the left string through α_s in the direction of α_r, i.e. the indices of $\alpha_s + \alpha_r, \alpha_s + 2\alpha_r, \ldots, \alpha_s + q\alpha_r$. In other words, this returns the sequence $[r_1, \ldots, r_q]$ where $\alpha_{r_i} = \alpha_s + i\alpha_r$ and $\alpha_s + (q+1)\alpha_r$ is not a root. The condition $\alpha_r \neq \pm\alpha_s$ must be satisfied.

`LeftStringLength(R, r, s)`

The largest p such that $\alpha_s - p\alpha_r$ is a root of the root datum R. The condition $\alpha_s \neq \pm\alpha_r$ must be satisfied.

`RightStringLength(R, r, s)`

The largest q such that $\alpha_s + q\alpha_r$ is a root of the root datum R. The condition $\alpha_s \neq \pm\alpha_r$ must be satisfied.

Example H103E21

```
> R := RootDatum("G2");
> Sum(R, 1, Negative(R,5));
10
> IsPositive(R, 10);
false
> Negative(R, 10);
4
> P := PositiveRoots(R);
> P[1] - P[5] eq -P[4];
true
```

`RootHeight(R, r)`

`CorootHeight(R, r)`

The height of the rth (co)root of the root datum R, i.e. the sum of the coefficients of α_r (respectively, $\alpha_r^\star$) with respect to the simple (co)roots.

`RootNorms(R)`

`CorootNorms(R)`

The sequence of squares of the lengths of the (co)roots of the root datum R.

`RootNorm(R, r)`

`CorootNorm(R, r)`

The square of the length of the rth (co)root of the root datum R.

`IsLongRoot(R, r)`

Returns `true` if, and only if, the rth root of the root datum R is long. This only makes sense for irreducible crystallographic root data. Note that for non-reduced root data, the roots which are not indivisible, are actually longer than the long ones.

`IsShortRoot(R, r)`

Returns `true` if, and only if, the rth root of the root datum R is short. This only makes sense for irreducible crystallographic root data.

`IsIndivisibleRoot(R, r)`

Returns `true` if, and only if, the rth root of the root system R is indivisible.

Example H103E22

Note that the Coxeter form is defined over the rationals. Since it is not possible to multiply a lattice element by a rational matrix, (co)roots must be coerced into a rational vector space first.

```
> R := RootDatum("G2");
> RootHeight(R, 5);
4
> F := CoxeterForm(R);
> v := VectorSpace(Rationals(),2) ! Root(R, 5);
> (v*F, v) eq RootNorm(R, 5);
true
> IsLongRoot(R, 5);
true
> LeftString(R, 1, 5);
[ 4, 3, 2 ]
> roots := Roots(R);
> for i in [1..3] do
>   RootPosition(R, roots[5]-i*roots[1]);
> end for;
4
3
2
```

`RootClosure(R, S)`

The closure in the root datum R of the set S of root indices. That is the indices of every root that can be written as a sum of roots with indices in S.

`AdditiveOrder(R)`

An additive order on the positive roots of the root datum R, ie. a sequence containing the numbers $1, \ldots, N$ in some order such that $\alpha_r + \alpha_s = \alpha_t$ implies t is between r and s. This is computed using the techniques of [Pap94]

`IsAdditiveOrder(R, Q)`

Returns `true` if, and only if, Q gives an additive order on a set of positive roots of R. Q must be a sequence of integers in the range $[1..N]$ with no gaps or repeats.

Example H103E23

```
> R := RootDatum("A5");
> a := AdditiveOrder(R);
> Position(a, 2);
6
> Position(a, 3);
10
> Position(a, Sum(R, 2, 3));
7
```

103.5.4 Weights

`WeightLattice(R)`

The weight lattice Λ of the root datum R. i.e. the λ in $\mathbf{Q}\Phi \leq \mathbf{Q} \otimes X$ such that $\langle \lambda, \alpha^\star \rangle \in \mathbf{Z}$ for every coroot $\alpha^\star$.

`CoweightLattice(R)`

The coweight lattice $\Lambda^\star$ of the root datum R, i.e. the $\lambda^\star$ in $\mathbf{Q}\Phi^\star \leq \mathbf{Q} \otimes Y$ such that $\langle \alpha, \lambda^\star \rangle \in \mathbf{Z}$ for every root α.

`FundamentalWeights(R)`

`Basis` MonStgElt *Default :* "*Standard*"

The fundamental weights $\lambda_1, \ldots, \lambda_n$ of the root datum R given as the rows of a matrix. This is the basis of the weight lattice Λ dual to the simple coroots, i.e. $\langle \lambda_i, \alpha_j^\star \rangle = \delta_{ij}$.

`FundamentalCoweights(R)`

`Basis` MonStgElt *Default :* "*Standard*"

The fundamental coweights $\lambda_1^\star, \ldots, \lambda_n^\star$ of the root datum R given as the rows of a matrix. This is the basis of the coweight lattice $\Lambda^\star$ dual to the simple roots, i.e. $\langle \alpha_i, \lambda_j^\star \rangle = \delta_{ij}$.

Example H103E24

```
> R := RootDatum("E6");
> WeightLattice(R);
Lattice of rank 6 and degree 6
Basis:
(4  3  5  6  4  2)
(3  6  6  9  6  3)
(5  6 10 12  8  4)
(6  9 12 18 12  6)
(4  6  8 12 10  5)
(2  3  4  6  5  4)
Basis Denominator: 3
> FundamentalWeights(R);
[ 4/3    1  5/3    2  4/3  2/3]
[   1    2    2    3    2    1]
[ 5/3    2 10/3    4  8/3  4/3]
[   2    3    4    6    4    2]
[ 4/3    2  8/3    4 10/3  5/3]
[ 2/3    1  4/3    2  5/3  4/3]
```

`IsDominant(R, v)`

`Basis` MonStgElt *Default :* "*Standard*"

Returns `true` if, and only if, v is a dominant weight for the root datum R, ie, a nonnegative integral linear combination of the fundamental weights.

`DominantWeight(R, v)`

`Basis` MonStgElt *Default :* "*Standard*"

The unique dominant weight in the same W-orbit as the weight v, where W is the Weyl group of the root datum R. The second value returned is a Weyl group element taking v to the dominant weight. The weight v can be given either as a vector or as a sequence representing the vector and is coerced into the weight lattice first.

`WeightOrbit(R, v)`

`Basis` MonStgElt *Default :* "*Standard*"

The W-orbit of the weight v as an indexed set, where W is the Weyl group of the root datum R. The first element in the orbit is always dominant. The second value returned is a sequence of Weyl group elements taking the dominant weight to the corresponding element of the orbit. The weight v can be given either as a vector or as a sequence representing the vector and is coerced into the weight lattice first.

Example H103E25

```
> R := RootDatum("B3");
> DominantWeight(R, [1,-1,0] : Basis:="Weight");
(1 0 0)
[ 2, 3, 2, 1 ]
> #WeightOrbit(R, [1,-1,0] : Basis:="Weight");
6
```

103.6 Building Root Data

`sub< R | a >`

The root subdatum of the root datum R generated by the roots $\alpha_{a_1}, \ldots, \alpha_{a_k}$ where $a = \{a_1, \ldots, a_k\}$ is a set of integers.

`sub< R | s >`

The root subdatum of the root datum R generated by the roots $\alpha_{s_1}, \ldots, \alpha_{s_k}$ where $s = [s_1, \ldots, s_k]$ is a *sequence* of integers. In this version the roots must be simple in the root subdatum (i.e. none of them may be a summand of another) otherwise an error is signalled. The simple roots will appear in the subdatum in the given order.

Example H103E26

```
> R := RootDatum("A4");
> PositiveRoots(R);
{@
    (1 0 0 0),
    (0 1 0 0),
    (0 0 1 0),
    (0 0 0 1),
    (1 1 0 0),
    (0 1 1 0),
    (0 0 1 1),
    (1 1 1 0),
    (0 1 1 1),
    (1 1 1 1)
@}
> s := sub< R | [6,1,4] >;
> s;
Root datum of type A3
> PositiveRoots(s);
{@
    (0 1 1 0),
    (1 0 0 0),
```

```
    (0 0 0 1),
    (1 1 1 0),
    (0 1 1 1),
    (1 1 1 1)
@}
> s := sub< R | [1,5] >;
Error: The given roots are not simple in a subdatum
> s := sub< R | {1,5} >;
> s;
Root datum of type A2
> PositiveRoots(s);
{@
    (1 0 0 0),
    (0 1 0 0),
    (1 1 0 0)
@}
```

`R1 subset R2`

Returns `true` if and only if the root datum R_1 is a subset of the root datum R_2. If true, returns an injection as sequence of roots as second return value.

`R1 + R2`

`DirectSum(R1, R2)`

The external direct sum of the root data R_1 and R_2. The full (co)root space of the result is the direct sum of the full (co)root spaces of R_1 and R_2.

`R1 join R2`

The internal direct sum of the root data R_1 and R_2. The root data must have the same full (co)root space, which will also be the full (co)root space of the result. The root data must have disjoint (co)root spaces.

Example H103E27

```
> R  := RootDatum("A1A1");
> R1 := sub<R|[1]>;
> R2 := sub<R|[2]>;
> R1 + R2;
Root datum of dimension 4 of type A1 A1
> R1 join R2;
R: Adjoint root datum of dimension 2 of type A1 A1
```

`DirectSumDecomposition(R)`

`IndecomposableSummands(R)`

Returns a sequence Q of irreducible root data, a root datum S which is the direct sum of the terms of Q, and an isogeny map $\phi : S \to R$. The root datum R must be semisimple. Note that a semisimple root datum R need not be a direct sum of simple root data, but it is isogenous to a direct sum of root data S.

Example H103E28

If the root datum in adjoint or simply connected, then it is a direct sum of simples. In this case we get $S = R$.

```
> R := RootDatum("A4B5" : Isogeny:="SC");
> Q, S := DirectSumDecomposition( R );
> R eq S;
true
> R eq Q[1] join Q[2];
true
```

The join of the summands of the direct sum decomposition is the original root datum again:

```
> R eq &join DirectSumDecomposition(R);
true
> R eq &+    DirectSumDecomposition(R);
false

> R1 := RootDatum("A3T2B4T3");
> R2 := RootDatum("T3G2T4BC3");
> R1 + R2;
Adjoint root datum of dimension 24 of type A3 B4 G2 BC3
> R1 join R2;
Root datum of dimension 12 of type A3 B4 G2 BC3
```

Here is an example of a semisimple root datum which is not a direct sum of simple subdata. Note that a simple root datum of type A_1 is either simply connected or adjoint.

```
> G<a,b>:=FundamentalGroup("A1A1");
> _,inj:=sub<G|a*b>;
> R:=RootDatum("A1A1":Isogeny:=inj);
> ad := RootDatum( "A1" : Isogeny:="Ad" );
> sc := RootDatum( "A1" : Isogeny:="SC" );
> IsIsomorphic( R, DirectSum(ad,ad) );
false
> IsIsomorphic( R, DirectSum(ad,sc) );
false
> IsIsomorphic( R, DirectSum(sc,sc) );
false
> Q, S := DirectSumDecomposition( R );
> R eq S;
```

```
false
```

`Dual(R)`

The dual of the root datum R, obtained by swapping the roots and coroots. The second value returned is the dual morphism from R to its dual.

`SimplyConnectedVersion(R)`

The simply connected version of the root datum R. If R is semisimple then the injection of the simply connected version into R is returned as the second value.

`AdjointVersion(R)`

The adjoint version of the root datum R. If R is semisimple then the projection from R to its adjoint version is returned as the second value.

`IndivisibleSubdatum(R)`

The root datum consisting of all indivisible roots of the root datum R.

`Radical(R)`

The radical of the root datum R, ie, the toral subdatum whose root (resp. coroot) space consists of the vectors perpendicular to every coroot (resp. root).

Example H103E29

An adjoint or simply connect root datum is always a direct sum of irreducible subdata. In these cases we take $S = R$.

```
> R1 := RootDatum("A5");
> R2 := RootDatum("B4");
> R := DirectSum(R1, Dual(R2));
> DirectSumDecomposition(R);
{
    Root datum of type A5 ,
    Root datum of type C4
}

> R := RootDatum("BC2");
> I := IndivisibleSubdatum(R); I;
I: Root datum of type B2
> I subset R;
true [ 1, 2, 3, 5, 7, 8, 9, 11 ]

> R := StandardRootDatum("A", 3);
> Radical(R);
Toral root datum of dimension 1
```

`TwistedRootDatum(R)`
`TwistedRootDatum(N)`

`Twist` ANY *Default :* 1

Create a twisted root datum from the root datum R, or from the semisimple root datum with Cartan name N. The twist may be specified in any of the following ways:

- An integer, specifying the order of the twist;
- A permutation, specifying the action of the primitive roots;
- A pair $< D, i >$, where D is a set of distinguished orbits as sets of integers, and i is the order of the Dynkin diagram symmetry;
- A pair $< \Gamma, Q >$, where Γ is the acting group, and Q is a sequence containing the permutation of the primitive roots for each of the generators of Γ;
- A homomorphism from Γ to the symmetric group whose order is the number of roots of R, describing how the acting group Γ acts on the roots.

Example H103E30

We construct a twisted root datum in a number of ways.

```
> S := TwistedRootDatum("D4" : Twist := 3);
> S;
S: Twisted adjoint root datum of dimension 4 of type 3D4,2

> R := RootDatum("A1A3");
> DynkinDiagram(R);

A1    1

A3    2 - 3 - 4
> S := TwistedRootDatum(R : Twist := Sym(4)!(2,4));
> S;
S: Twisted adjoint root datum of dimension 4 of type 2(A1 A3)4,3

> S := TwistedRootDatum("A4" : Twist := <{{1,4},{2,3}}, 2>);
> S;
S: Twisted adjoint root datum of dimension 4 of type 2A4,2

> R := RootDatum("E6" : Isogeny := "SC");
> DynkinDiagram(R);

E6    1 - 3 - 4 - 5 - 6
              |
              2
> S := TwistedRootDatum(R : Twist := <Sym(2) ,[ Sym(6)!(1,6)(3,5) ]>);
> S;
```

```
S: Twisted simply connected root datum of dimension 6 of type 2E6,4

> R := RootDatum("D4");
> DynkinDiagram(R);

D4    3
      /
1 - 2
      \
       4
> Gamma := Sym(3);
> Gamma.1, Gamma.2;
(1, 2, 3)
(1, 2)
> S := TwistedRootDatum(R : Twist := <Gamma, [ Sym(4) | (1,3,4), (1,4) ]>);
> S;
S: Twisted adjoint root datum of dimension 4 of type 6D4,2

> R := RootDatum("A2");
> DynkinDiagram(R);

A2    1 - 2
> Roots(R);
{@
    (1 0),
    (0 1),
    (1 1),
    (-1  0),
    ( 0 -1),
    (-1 -1)
@}
> S6 := Sym(#Roots(R));
> phi := hom<Sym(2) -> S6 | S6!(1,2)(4,5)>;
> S := TwistedRootDatum(R : Twist := phi);
```

`UntwistedRootDatum(R)`

`SplitRootDatum(R)`

The split version of the (twisted) root datum R.

103.7 Morphisms of Root Data

Morphisms are currently only defined for split root data. Let $R_i = (X_i, \Phi_i, Y_i, \Phi_i^\star)$ be a root datum for $i = 1, 2$. A *morphism* of root data $\phi : R_1 \to R_2$ consists of a pair of $\mathbf{Z}$-linear maps $\phi_X : X_1 \to X_2$ and $\Phi_Y : Y_1 \to Y_2$ satisfying

1. $\phi_X(\Phi_1) \subseteq \Phi_2 \cup \{0\}$; and
2. $\phi_Y(\alpha^\star) = \phi_X(\alpha)^\star$ (with the convention that $0^\star = 0$).

A *fractional morphism* is similar, except that it consists of $\mathbf{Q}$-linear maps on the (co)root spaces $X_1 \otimes \mathbf{Q} \to X_2 \otimes \mathbf{Q}$ and $Y_1 \otimes \mathbf{Q} \to Y_2 \otimes \mathbf{Q}$. The main examples of fractional morphisms are isogeny maps (Section 103.1.6). A *dual morphism* is similar, except that the maps are $X_1 \to Y_2$ and $Y_1 \to X_2$. This is clearly equivalent to a morphism from R_1 to the dual of R_2. Finally we define a *dual fractional morphism* in the obvious way.

A (fractional) morphism $\phi : R_1 \to R_2$ also stores a sign corresponding to each simple root of R_1. This has no effect on the action of ϕ on roots or coroots, but does effect the definition of the corresponding homomorphisms of Lie algebras and groups of Lie type.

`hom< R -> S | phiX, phiY >`

Construct a (fractional) morphism $\phi : R \to S$ of root data with the given linear maps or matrices `phiX` and `phiY` for the action of ϕ on X_1 and Y_1.

`hom< R -> S | Q >`

Construct a (fractional) morphism of root data $R \to S$ with the given sequence of root images. The sequence Q must have length $2N$ and consist of elements in the range $[0, \ldots, 2M]$, where N is the number of positive roots of R and M is the number of positive roots of S. The domain R must be semisimple.

`Morphism(R, S, phiX, phiY)`

`Check` BoolElt *Default :* `true`

Construct a (fractional) morphism $\phi : R \to S$ of root data with the given linear maps or matrices `phiX` and `phiY` for the action of ϕ on X_1 and Y_1. The domain R must be semisimple.

If `Check` is set to `false`, the function does not check that the maps send (co)roots to (co)roots. This function is the same as the constructor `hom`, except for the optional parameter.

`Morphism(R, S, Q)`

`Check` BoolElt *Default :* `true`

Construct a (fractional) morphism of root data $R \to S$ with the given sequence of root images. The sequence Q must have length $2N$ and consist of elements in the range $[0, \ldots, 2M]$, where N is the number of positive roots of R and M is the number of positive roots of S. The domain R must be semisimple.

If `Check` is set to `false`, the function does not check that the maps send (co)roots to (co)roots. This function is the same as the constructor `hom`, except for the optional parameter.

`DualMorphism(R, S, phiX, phiY)`

`Check` BoolElt *Default :* `true`

Construct a (fractional) dual morphism of root data $R \to S$ with the given linear maps or matrices of linear maps. If `Check` is set to `false`, the function does not check that the maps send (co)roots to (co)roots.

`DualMorphism(R, S, Q)`

`Check` BoolElt *Default :* `true`

Construct a (fractional) dual morphism of root data $R \to S$ with the given sequence of root images. The sequence Q must have length $2N$ and consist of elements in the range $[0, \ldots, 2M]$, where N is the number of positive roots of R and M is the number of positive roots of S. The domain R must be semisimple. If `Check` is set to `false`, the function does not check that the maps send (co)roots to (co)roots.

`RootImages(phi)`

The indices of the root images of the (dual) (fractional) morphism ϕ.

`RootPermutation(phi)`

The indices of the root images of the automorphism ϕ.

`IsIsogeny(phi)`

Returns `true` if the morphism ϕ is an isogeny, ie, ϕ_Y is onto with finite kernel.

`IdentityMap(R)`

`IdentityAutomorphism(R)`

The identity morphism $R \to R$.

Example H103E31

We construct the fractional morphism from the standard root datum of type A_3 onto the adjoint root datum of type A_3. This will allow us to construct the algebraic projection $GL_4 \to PGL_4$ in Section 109.11.

```
> RGL := StandardRootDatum( "A", 3 );
> RPGL := RootDatum( "A3" );
> A := VerticalJoin( SimpleRoots(RGL), Vector([Rationals()|1,1,1,1]) )^-1 *
>   VerticalJoin( SimpleRoots(RPGL), Vector([Rationals()|0,0,0]) );
> B := VerticalJoin( SimpleCoroots(RGL), Vector([Rationals()|1,1,1,1]) )^-1 *
>   VerticalJoin( SimpleCoroots(RPGL), Vector([Rationals()|0,0,0]) );
> phi := hom< RGL -> RPGL | A, B >;
> v := Coroot(RGL,1);
> v; phi(v);
( 1 -1  0  0)
( 2 -1  0 )
```

103.8 Constants Associated with Root Data

In this section functions for a number of constants associated with root data will be described. These constants are needed to define Lie algebras and groups of Lie type. The notation of [Car72] will be used, except that the constants are defined for right actions rather than left actions [CMT04].

`ExtraspecialPairs(R)`

The sequence of extraspecial pairs of the root datum R (see [Car72, page 58]). That is the sequence $[(r_i, s_i)]_{i=1}^{N-n}$ where r_i is minimal such that $\alpha_{r_i} + \alpha_{s_i} = \alpha_{i+n}$ (n is the rank of R and N is the number of positive roots).

`NumExtraspecialPairs(R)`

The number of extraspecial pairs of the root datum R. This function doesn't actually compute the extraspecial pairs, thus is much more efficient than calling `#ExtraspecialPairs(R)` in case extraspecial pairs are not yet computed.

`ExtraspecialPair(R,r)`

The extraspecial pair of the rth root in the root datum R. That is the pair (s, t) where s is minimal such that $\alpha_s + \alpha_t = \alpha_r$.

`ExtraspecialSigns(R)`

Return the sequence of extraspecial signs of the root datum R.

`LieConstant_p(R, r, s)`

The constant p_{rs} for the root datum R, i.e. the largest p such that $\alpha_s - p\alpha_r$ is a root. This is the same as `LeftStringLength`. The condition $\alpha_s \neq \pm\alpha_r$ must be satisfied.

`LieConstant_q(R, r, s)`

The constant q_{rs} for the root datum R, i.e. the largest q such that $\alpha_s + q\alpha_r$ is a root. This is the same as `RightStringLength`. The condition $\alpha_s \neq \pm\alpha_r$ must be satisfied.

`CartanInteger(R, r, s)`

The Cartan integer $\langle \alpha_r, \alpha_s^\star \rangle$ for the root datum R.

`LieConstant_N(R, r, s)`

The Lie algebra structure constant N_{rs} for the root datum R. The condition $\alpha_s \neq \pm\alpha_r$ must be satisfied.

`LieConstant_epsilon(R, r, s)`

The constant $\epsilon_{rs} = \text{Sign}(N_{rs})$ for the root datum R. The condition $\alpha_s \neq \pm\alpha_r$ must be satisfied.

`LieConstant_M(R, r, s, i)`

The constant $M_{rsi} = \frac{1}{i!} N_{s_0 r} \cdots N_{s_{i-1} r}$ where $\alpha_{s_i} = i\alpha_r + \alpha_s$ for the root datum R. The condition $\alpha_s \neq \pm\alpha_r$ must be satisfied.

`LieConstant_C(R, i, j, r, s)`

The Lie group structure constant C_{ijrs} for the root datum R. The conditions $\alpha_s \neq \pm\alpha_r$ and $\alpha_r + \alpha_s \in \Phi$ must be satisfied.

`LieConstant_eta(R, r, s)`

The constant

$$\eta_{rs} = (-1)^{p_{rs}} \frac{\epsilon_{r,s-pr} \cdots \epsilon_{r,s-r}}{\epsilon_{r,s-pr} \cdots \epsilon_{r,s+(q-p-1)r}}$$

for the root datum R. The condition $\alpha_s \neq \pm\alpha_r$ must be satisfied.

`StructureConstants(R)`

The Lie algebra structure constants for the reductive Lie algebra with root datum R in the sparse format described in Section 106.2.

Example H103E32

The code below verifies some standard formulas in the root datum of type F_4:

```
> R := RootDatum("F4");
> N := NumPosRoots(R);
> r := Random([1..N]);
> s := Random([1..r-1] cat [r+1..r+N-1] cat [r+N+1..2*N]);
```

1. Agreement of the Cartan matrix with the Cartan integers.

```
> C := CartanMatrix(R);
> C[2,3] eq CartanInteger(R,2,3);
true
```

2. p_{rs} is the length of the left string through α_s in the direction of α_r.

```
> LieConstant_p(R,r,s) eq #LeftString(R,r,s);
true
```

3. q_{rs} is the length of the right string through α_s in the direction of α_r.

```
> LieConstant_q(R,r,s) eq #RightString(R,r,s);
true
```

4. $\langle \alpha_s, \alpha_r^\star \rangle = p_{rs} - q_{rs}$.

```
> CartanInteger(R,s,r) eq
> LieConstant_p(R,r,s) - LieConstant_q(R,r,s);
```

```
true
```

5. $N_{rs} = \epsilon_{rs}(p_{rs} + 1)$.

```
> LieConstant_N(R,r,s) eq
> LieConstant_epsilon(R,r,s) * (LieConstant_p(R,r,s) + 1);
true
```

103.9 Related Structures

In this section functions for creating other structures from a root datum are briefly listed. See the appropriate chapters of the Handbook for more details.

`RootSystem(R)`

The root system corresponding to the root datum R. See Chapter 102.

`CoxeterGroup(GrpFPCox, R)`

The (split) Coxeter group with root datum R. See Chapter 104. The braid group and pure braid group can be computed from the Coxeter group using the commands described in Section 104.12.

`CoxeterGroup(R)`

`CoxeterGroup(GrpPermCox, R)`

The permutation Coxeter group with root datum R. See Chapter 104.

`ReflectionGroup(R)`

`CoxeterGroup(GrpMat, R)`

The reflection group of the root datum R. See Chapter 105.

`LieAlgebraHomorphism(phi,k)`

The homomorphism of reductive Lie algebras over the ring k corresponding to the root datum morphism ϕ. See Chapter 106.

`LieAlgebra(R, k)`

The reductive Lie algebra over the ring k with root datum R. See Chapter 106.

`GroupOfLieType(R, k)`

The group of Lie type over the ring k with root datum R. See Chapter 109.

`GroupOfLieTypeHomomorphism(phi, k)`

The algebraic homomorphism of groups of Lie type over the ring k corresponding to the root datum morphism ϕ. See Chapter 109.

Example H103E33

```
> R := RootDatum("b3");
> SemisimpleType(LieAlgebra(R, Rationals()));
B3
> #CoxeterGroup(R);
48
> GroupOfLieType(R, Rationals());
$: Group of Lie type B3 over Rational Field
```

103.10 Bibliography

[**Bou68**] N. Bourbaki. *Éléments de mathématique. Fasc. XXXIV. Groupes et algèbres de Lie. Chapitre IV: Groupes de Coxeter et systèmes de Tits. Chapitre V: Groupes engendrés par des réflexions. Chapitre VI: Systèmes de racines.* Hermann, Paris, 1968.

[**Car72**] Roger W. Carter. *Simple groups of Lie type.* John Wiley & Sons, London-New York-Sydney, 1972. Pure and Applied Mathematics, Vol. 28.

[**Car93**] Roger W. Carter. *Finite groups of Lie type.* John Wiley & Sons, Chichester, 1993. Conjugacy classes and complex characters, Reprint of the 1985 original, A Wiley-Interscience Publication.

[**CHM08**] Arjeh M. Cohen, Sergei Haller, and Scott H. Murray. Computing in unipotent and reductive algebraic groups. *LMS J. Comput. Math.*, 11:343–366, 2008.

[**CMT04**] Arjeh M. Cohen, Scott H. Murray, and D. E. Taylor. Computing in groups of Lie type. *Math. Comp.*, 73(247):1477–1498, 2004.

[**Dem70**] M. Demazure. Données radicielles. In *Schémas en Groupes III, Exposeé XXI*, volume 153 of *Lecture Notes in Mathematics*, pages 85–155. Springer-Verlag, Berlin, 1970.

[**Hal05**] Sergei Haller. *Computing Galois Cohomology and Forms of Linear Algebraic Groups.* Phd thesis, Technical University of Eindhoven, 2005.

[**Pap94**] Paolo Papi. A characterization of a special ordering in a root system. *Proc. Amer. Math. Soc.*, 120(3):661–665, 1994.

[**Sat71**] I. Satake. *Classification theory of semi-simple algebraic groups.* Marcel Dekker Inc., New York, 1971. With an appendix by M. Sugiura, Notes prepared by Doris Schattschneider, Lecture Notes in Pure and Applied Mathematics, 3.

[**Sch69**] Doris J. Schattschneider. On restricted roots of semi-simple algebraic groups. *J. Math. Soc. Japan*, 21:94–115, 1969.

104 COXETER GROUPS

104.1 Introduction 3139

104.1.1 The Normal Form for Words . . 3140

104.2 Constructing Coxeter Groups 3140

CoxeterGroup(GrpFPCox, N) 3140
CoxeterGroup(GrpPermCox, N) 3140
CoxeterGroup(N) 3140
IrreducibleCoxeter Group(GrpFPCox, X, n) 3140
CoxeterGroup(GrpFPCox, M) 3141
CoxeterGroup(GrpPermCox, M) 3141
CoxeterGroup(M) 3141
CoxeterGroup(GrpFPCox, G) 3141
CoxeterGroup(GrpPermCox, G) 3141
CoxeterGroup(G) 3141
CoxeterGroup(GrpFPCox, C) 3141
CoxeterGroup(GrpPermCox, C) 3141
CoxeterGroup(C) 3141
CoxeterGroup(GrpFPCox, D) 3141
CoxeterGroup(GrpPermCox, D) 3141
CoxeterGroup(D) 3141
CoxeterGroup(GrpFPCox, R) 3143
CoxeterGroup(GrpPermCox, R) 3143
CoxeterGroup(R) 3143
CoxeterGroup(A, B) 3143

104.3 Converting Between Types of Coxeter Group 3143

CoxeterGroup(GrpFPCox, W) 3144
CoxeterGroup(GrpFPCox, W) 3144
CoxeterGroup(GrpPermCox, W) 3144
CoxeterGroup(GrpPermCox, W) 3144
ReflectionGroup(W) 3144
CoxeterGroup(GrpMat, W) 3144
ReflectionGroup(W) 3145
CoxeterGroup(GrpMat, W) 3145
CoxeterGroup(GrpFP, W) 3145
CoxeterGroup(GrpFP, W) 3145
CoxeterGroup(GrpFP, W) 3145
CoxeterGroup(GrpPerm, W) 3145
CoxeterGroup(GrpPerm, W) 3145
CoxeterGroup(GrpPerm, W) 3145

104.4 Operations on Coxeter Groups3146

IsIsomorphic(W1, W2) 3146
IsCoxeterIsomorphic(W1, W2) 3146
IsCartanEquivalent(W1, W2) 3146
RootSystem(W) 3146
RootDatum(W) 3147
CartanName(W) 3147
CoxeterDiagram(W) 3147
DynkinDiagram(W) 3147
CoxeterMatrix(W) 3147
CoxeterGraph(W) 3148
CartanMatrix(W) 3148
DynkinDigraph(W) 3148
Rank(W) 3148
NumberOfGenerators(W) 3148
NumberOfPositiveRoots(W) 3148
NumPosRoots(W) 3148
Dimension(W) 3148
ConjugacyClasses(W) 3148
FundamentalGroup(W) 3148
IsogenyGroup(W) 3148
CoisogenyGroup(W) 3149
BasicDegrees(W) 3149
BasicCodegrees(W) 3149
BruhatLessOrEqual(x, y) 3149
BruhatDescendants(x) 3149
BruhatDescendants(X) 3150

104.5 Properties of Coxeter Groups 3150

IsFinite(W) 3150
IsAffine(W) 3150
IsHyperbolic(W) 3150
IsCompactHyperbolic(W) 3150
IsIrreducible(W) 3151
IsSemisimple(W) 3151
IsCrystallographic(W) 3151
IsSimplyLaced(W) 3151

104.6 Operations on Elements . . 3152

3152
Length(w) 3152
CoxeterLength(w) 3152
LongestElement(W) 3152
CoxeterElement(W) 3152
CoxeterNumber(W) 3152
LeftDescentSet(W, w) 3153
RightDescentSet(W, w) 3153

104.7 Roots, Coroots and Reflections3154

104.7.1 Accessing Roots and Coroots . . 3154

RootSpace(W) 3154
CorootSpace(W) 3154
SimpleRoots(W) 3154
SimpleCoroots(W) 3154
NumberOfPositiveRoots(W) 3155
NumPosRoots(W) 3155
Roots(W) 3155
Coroots(W) 3155
PositiveRoots(W) 3155
PositiveCoroots(W) 3155
Root(W, r) 3155
Coroot(W, r) 3155
RootPosition(W, v) 3155
CorootPosition(W, v) 3155
HighestRoot(W) 3156
HighestLongRoot(W) 3156
HighestShortRoot(W) 3156

CoxeterForm(W) 3157
DualCoxeterForm(W) 3157
AdditiveOrder(W) 3157
104.7.2 Operations and Properties for Root and Coroot Indices 3157
Sum(W, r, s) 3157
IsPositive(W, r) 3157
IsNegative(W, r) 3157
Negative(W, r) 3157
LeftString(W, r, s) 3157
RightString(W, r, s) 3158
LeftStringLength(W, r, s) 3158
RightStringLength(W, r, s) 3158
RootHeight(W, r) 3158
CorootHeight(W, r) 3158
RootNorms(W) 3158
CorootNorms(W) 3158
RootNorm(W, r) 3158
CorootNorm(W, r) 3158
IsLongRoot(W, r) 3159
IsShortRoot(W, r) 3159
104.7.3 Weights 3159
WeightLattice(W) 3159
CoweightLattice(W) 3159
FundamentalWeights(W) 3159
FundamentalCoweights(W) 3159
IsDominant(R, v) 3160
DominantWeight(W, v) 3160
WeightOrbit(W, v) 3160

104.8 Reflections 3160

IsReflection(w) 3160
Reflections(W) 3160
SimpleReflections(W) 3161
SimpleReflectionPermutations(W) 3161
Reflection(W, r) 3161
ReflectionPermutation(W, r) 3161
SimpleReflectionMatrices(W) 3161
SimpleCoreflectionMatrices(W) 3161
ReflectionMatrices(W) 3161
CoreflectionMatrices(W) 3161
ReflectionMatrix(W, r) 3161
CoreflectionMatrix(W, r) 3161
ReflectionWords(W) 3162
ReflectionWord(W, r) 3162

104.9 Reflection Subgroups . . . 3162

ReflectionSubgroup(W, a) 3162
ReflectionSubgroup(W, s) 3163
StandardParabolicSubgroup(W, J) 3163
IsReflectionSubgroup(W, H) 3163
IsParabolicSubgroup(W, H) 3163
IsStandardParabolicSubgroup(W, H) 3163
Overgroup(H) 3163
Overdatum(H) 3163
LocalCoxeterGroup(H) 3163
Transversal(W, H) 3164
TransversalWords(W, H) 3164
TransversalElt(W, H, x) 3164
TransversalElt(W, x, H) 3165
TransversalElt(W, H, x, J) 3165
Transversal(W, J) 3165
Transversal(W, J, L) 3165
Transversal(W, J, K) 3165
DirectProduct(W1, W2) 3165
Dual(W) 3165

104.10 Root Actions 3166

RootGSet(W) 3166
CorootGSet(W) 3166
RootAction(W) 3166
CorootAction(W) 3166
ReflectionGroup(W) 3167
CoreflectionGroup(W) 3167

104.11 Standard Action 3167

StandardAction(W) 3167
StandardActionGroup(W) 3167

104.12 Braid Groups 3168

BraidGroup(W) 3168
PureBraidGroup(W) 3168

104.13 W-graphs 3169

SetVerbose("WGraph", v) 3169
Mij2EltRootTable(seq) 3170
Name2Mij(name) 3170
Partition2WGtable(pi) 3171
WGtable2WG(table) 3171
TestWG(W,wg) 3171
TestWG(tp,wg) 3171
WGelement2WGtable(g,K) 3172
GetCells(wg) 3172
InduceWG(W,wg,seq) 3172
InduceWGtable(J, table, W) 3172
IsWGsymmetric(dwg) 3172
MakeDirected(uwg) 3172
TestHeckeRep(W,r) 3172
WG2GroupRep(wg) 3173
WG2HeckeRep(W,wg) 3173
WGidealgens2WGtable(dgens,K) 3173
WriteWG(file,uwg) 3173
WriteWG(file,dwg) 3173

104.14 Related Structures 3174

CoxeterGroup(GrpFP, W) 3174
Presentation(W) 3174
ReflectionGroup(W) 3174
CoxeterGroup(GrpMat, W) 3174
LieAlgebra(W, R) 3174
GroupOfLieType(W, R) 3174

104.15 Bibliography 3174

Chapter 104
COXETER GROUPS

104.1 Introduction

This chapter describes Magma functions for computing with Coxeter groups. A *Coxeter system* is a group G with finite generating set $S = \{s_1, \ldots, s_n\}$, defined by relations $s_i^2 = 1$ for $i = 1, \ldots, n$ and

$$s_i s_j s_i \cdots = s_j s_i s_j \cdots$$

for $i, j = 1, \ldots, n$ with $i < j$, where each side of this relation has length $m_{ij} \geq 2$. Traditionally, $m_{ij} = \infty$ signifies that the corresponding relation is omitted but, for technical reasons, $m_{ij} = 0$ is used in MAGMA instead. The group G is called a *Coxeter group* and S is called the set of *Coxeter generators*. Since every group in MAGMA has a preferred generating set, no distinction is made between a Coxeter system and its Coxeter group. See [Bou68] for more details on the theory of Coxeter groups.

The *rank* of the Coxeter system is $n = |S|$. A Coxeter system is said to be *reducible* if there is a proper subset I of $\{1, \ldots, n\}$ such that $m_{ij} = 2$ or $m_{ji} = 2$ whenever $i \in I$ and $j \notin I$. In this case, G is an (internal) direct product of the Coxeter subgroups $W_I = \langle s_i \mid i \in I \rangle$ and $W_{I^c} = \langle s_i \mid i \notin I \rangle$. Note that an *irreducible* Coxeter group may still be a nontrivial direct product of abstract subgroups (for example, $W(G_2) \cong S_2 \times S_3$). Two Coxeter groups are *Coxeter isomorphic* if there is a group isomorphism between them which takes Coxeter generators to Coxeter generators. In other words, the two groups are the same modulo renumbering of the generators.

MAGMA provides three methods for working with Coxeter groups:

1. As a finitely presented group with the standard presentation given above. These groups have type `GrpFPCox`. See Chapter 75 for general functions for finitely presented groups.
2. As a permutation group acting on the roots of the root system. Clearly the group must be finite. These groups have type `GrpPermCox`. See Chapter 63 for general functions for permutation groups.
3. As a reflection group, i.e. a matrix group generated by reflections. These groups have the same type as general matrix groups (`GrpMat`). They can be distinguished with the `IsReflectionGroup` function.

The first two methods are described in this chapter. The third is described in Chapter 105.

A permutation Coxeter group always has an underlying root system or root datum, and so many commands involving roots also work for these groups. A finitely presented Coxeter group does not have such an underlying structure.

The code for Coxeter groups as permutation groups was originally modelled on the corresponding part of the Chevie package of GAP [GHL+96] by Meinholf Geck, Frank Lübeck, Jean Michel and Götz Pfeiffer.

104.1.1 The Normal Form for Words

Every element w of a Coxeter group W can be written as a word

$$w = r_1 r_2 \cdots r_l$$

with each r_i in S. A *reduced expression* for w is such a word with l minimal; in this case, l is defined to be the *length* of w.

An ordering on words in S is obtained by taking the *lexicographic (alphabetic) order* induced by the existing ordering on S. The *normal form* for w in W is the smallest reduced expression for w with respect to this ordering. Algorithms for efficiently computing this normal form have been developed and implemented by R. B. Howlett. These algorithms are based on the concept of a minimal root [Bri98, BH93].

The main difference of the category of Coxeter groups (`GrpFPCox`) from the category of finitely presented groups (`GrpFP`) is that that all words are automatically put into this normal form. In particular, this means that two words are equal if, and only if, they are equal as group elements. Coxeter groups can also be constructed in the category `GrpFP` if the user wishes to avoid automatic normalisation of elements (see Section 104.3).

104.2 Constructing Coxeter Groups

It is possible to specify the category `GrpFPCox` or `GrpPermCox` when constructing a Coxeter group. If the category is not specified, then a `GrpPermCox` is returned for finite groups and a `GrpFPCox` is returned for infinite groups. If the category `GrpPermCox` is specified for an infinite group, an error is signalled.

`CoxeterGroup(GrpFPCox, N)`

`CoxeterGroup(GrpPermCox, N)`

`CoxeterGroup(N)`

The finite or affine Coxeter group with Cartan name given by the string N (see Section 101.6).

`IrreducibleCoxeterGroup(GrpFPCox, X, n)`

The finite or affine irreducible Coxeter group with Cartan name X_n, or $I_2(n)$ if $X =$`"I"` (see Section 101.6).

Example H104E1

```
> CoxeterGroup(GrpFPCox, "B3");
Coxeter group: Finitely presented group on 3 generators
Relations
    $.1 * $.2 * $.1 = $.2 * $.1 * $.2
    $.1 * $.3 = $.3 * $.1
    ($.2 * $.3)^2 = ($.3 * $.2)^2
```

```
    $.1^2 = Id($)
    $.2^2 = Id($)
    $.3^2 = Id($)
> CoxeterGroup("A2B2");
Coxeter group: Permutation group acting on a set of cardinality 14
Order = 48 = 2^4 * 3
    (1, 8)(2, 5)(9, 12)
    (1, 5)(2, 9)(8, 12)
    (3, 10)(4, 6)(11, 13)
    (3, 7)(4, 11)(10, 14)
```

```
CoxeterGroup(GrpFPCox, M)
CoxeterGroup(GrpPermCox, M)
CoxeterGroup(M)
```

The Coxeter group with Coxeter matrix M (see Chapter 101).

```
CoxeterGroup(GrpFPCox, G)
CoxeterGroup(GrpPermCox, G)
CoxeterGroup(G)
```

The Coxeter group with Coxeter graph G (see Chapter 101).

```
CoxeterGroup(GrpFPCox, C)
CoxeterGroup(GrpPermCox, C)
CoxeterGroup(C)
```

The Coxeter group with Cartan matrix C (see Chapter 101).

```
CoxeterGroup(GrpFPCox, D)
CoxeterGroup(GrpPermCox, D)
CoxeterGroup(D)
```

The Coxeter group with Dynkin digraph D (see Chapter 101).

Example H104E2

```
> M := SymmetricMatrix([ 1, 4,1, 3,4,1 ]);
> G<a,b,c> := CoxeterGroup(M);
> G;
Coxeter group: Finitely presented group on 3 generators
Relations
    (a * b)^2 = (b * a)^2
    a * c * a = c * a * c
    (b * c)^2 = (c * b)^2
    a^2 = Id($)
    b^2 = Id($)
    c^2 = Id($)
> M := SymmetricMatrix([ 1, 3,1, 2,3,1 ]);
> G<a,b,c> := CoxeterGroup(M);
> G;
Coxeter group: Permutation group G acting on a set of cardinality 12
Order = 24 = 2^3 * 3
    (1, 7)(2, 4)(5, 6)(8, 10)(11, 12)
    (1, 4)(2, 8)(3, 5)(7, 10)(9, 11)
    (2, 5)(3, 9)(4, 6)(8, 11)(10, 12)
> G<a,b,c> := CoxeterGroup(GrpFPCox, M);
> G;
Coxeter group: Finitely presented group on 3 generators
Relations
    a * b * a = b * a * b
    a * c = c * a
    b * c * b = c * b * c
    a^2 = Id($)
    b^2 = Id($)
    c^2 = Id($)
```

Note that a Coxeter group does not have a unique Cartan matrix.

```
> C := CartanMatrix("G2");
> W := CoxeterGroup(GrpFPCox, C);
>  CartanMatrix(W);
>> CartanMatrix(W);
               ^
Runtime error in 'CartanMatrix': Bad argument types
Argument types given: GrpFPCox
```

`CoxeterGroup(GrpFPCox, R)`

`CoxeterGroup(GrpPermCox, R)`

`CoxeterGroup(R)`

The finite Coxeter group with root system or root datum R (see Chapters 102 and 103).

`CoxeterGroup(A, B)`

The permutation Coxeter group with roots given by the rows of the matrix A and coroots given by the rows of the matrix B. The matrices A and B must have the following properties:

1. A and B must have same number of rows and the same number of columns; they must be defined over the same field, which must be the rational field, a number field, or a cyclotomic field; the entries must be real;
2. the number of columns must be at least the number of rows; and
3. AB^t must be the Cartan matrix of a finite Coxeter group.

Example H104E3

```
> R := RootDatum("A3" : Isogeny := 2);
> CoxeterGroup(R);
Coxeter group: Permutation group acting on a set of cardinality 12
Order = 24 = 2^3 * 3
    (1, 7)(2, 4)(5, 6)(8, 10)(11, 12)
    (1, 4)(2, 8)(3, 5)(7, 10)(9, 11)
    (2, 5)(3, 9)(4, 6)(8, 11)(10, 12)
```

104.3 Converting Between Types of Coxeter Group

In this section, we describe functions for converting between the various descriptions of Coxeter groups available in Magma.

Since a finitely presented Coxeter group W does not come with an in-built reflection representation, the optional parameters A, B, and C can be used to specify the representation. They are respectively the matrix whose rows are the simple roots, the matrix whose rows are the simple coroots, and the Cartan matrix. These must have the following properties:

1. A and B must have same number of rows and the same number of columns; they must be defined over the same field, which must be the rational field, a number field, or a cyclotomic field; the entries must be real;
2. the number of columns must be at least the number of rows; and
3. $C = AB^t$ must be a Cartan matrix for W.

It is not necessary to specify all three matrices, since any two of them will determine the third. If these matrices are not given, the default is to take A to be the identity and to take C to be the standard Cartan matrix described in Section 101.4.

`CoxeterGroup(GrpFPCox, W)`

The finitely presented Coxeter group W' isomorphic to the permutation Coxeter group W, together with the isomorphism $W \to W'$.

`CoxeterGroup(GrpFPCox, W)`

The finitely presented Coxeter group W' isomorphic to the real reflection group W (see Chapter 105).

`CoxeterGroup(GrpPermCox, W)`

`A`	MTRX	*Default :*
`B`	MTRX	*Default :*
`C`	MTRX	*Default :*

The permutation Coxeter group W' isomorphic to the finitely presented Coxeter group W, together with the isomorphism $W \to W'$. If W is infinite, an error is flagged.

`CoxeterGroup(GrpPermCox, W)`

The permutation Coxeter group W' isomorphic to the real reflection group W, together with the isomorphism $W \to W'$ (see Chapter 105). If W is infinite, an error is flagged.

Example H104E4

```
> W<a,b> := CoxeterGroup(GrpFPCox, "G2");
> Wp, h := CoxeterGroup(GrpPermCox, W);
> a*b;
a * b
> h(a*b);
(1, 11, 12, 7, 5, 6)(2, 4, 3, 8, 10, 9)
```

`ReflectionGroup(W)`

`CoxeterGroup(GrpMat, W)`

`A`	MTRX	*Default :*
`B`	MTRX	*Default :*
`C`	MTRX	*Default :*

A reflection group W' of the Coxeter group W, together with the isomorphism $W \to W'$.

`ReflectionGroup(W)`

`CoxeterGroup(GrpMat, W)`

The reflection group W' isomorphic to the permutation Coxeter group W, together with the isomorphism $W \to W'$. There are no optional parameters A, B, and C in this case because every permutation Coxeter group has a root system, and this determines the reflection representation.

Example H104E5

```
> W<a,b,c> := CoxeterGroup(GrpFPCox, "B3");
> G, h := CoxeterGroup(GrpMat, W);
> a*b; h(a*b);
a * b
[-1 -1  0]
[ 1  0  0]
[ 0  1  1]
```

`CoxeterGroup(GrpFP, W)`

The finitely presented group W' isomorphic to the finitely presented Coxeter group W, together with the isomorphism $W \to W'$.

`CoxeterGroup(GrpFP, W)`

The finitely presented group W' isomorphic to the permutation Coxeter group W, together with the isomorphism $W \to W'$.

`CoxeterGroup(GrpFP, W)`

The finitely presented group W' isomorphic to the real reflection group W, together with the isomorphism $W \to W'$ (see Chapter 105).

`CoxeterGroup(GrpPerm, W)`

The permutation group W' isomorphic to the finitely presented Coxeter group W, together with the isomorphism $W \to W'$. If W is infinite, an error is flagged.

`CoxeterGroup(GrpPerm, W)`

The permutation group W' isomorphic to the permutation Coxeter group W, together with the isomorphism $W \to W'$.

`CoxeterGroup(GrpPerm, W)`

The permutation group W' isomorphic to the real reflection group W, together with the isomorphism $W \to W'$ (see Chapter 105). If W is infinite, an error is flagged.

104.4 Operations on Coxeter Groups

See Chapter 75 for general functions for finitely presented groups, or Chapter 63 for general functions for permutation groups.

`IsIsomorphic(W1, W2)`

Returns `true` if, and only if, W_1 and W_2 are isomorphic as abstract groups. This is only implemented for permutation Coxeter groups.

`IsCoxeterIsomorphic(W1, W2)`

Tests if W_1 and W_2 are isomorphic as Coxeter systems. If `true`, a sequence giving the permutation of the generators which takes W_1 to W_2 is also returned.

`IsCartanEquivalent(W1, W2)`

Returns `true` if and only if the crystallographic Coxeter groups W_1 and W_2 have Cartan equivalent Cartan matrices. This only makes sense for permutation Coxeter groups.

Example H104E6

```
> W1 := CoxeterGroup(GrpFPCox, "B4");
> W2 := CoxeterGroup(GrpFPCox, "C4");
> IsCoxeterIsomorphic(W1, W2);
true [ 1, 2, 3, 4 ]
```

An example of abstractly isomorphic Coxeter groups whose Coxeter systems not isomorphic:

```
> W1 := CoxeterGroup("G2");
> W2 := CoxeterGroup("A1A2");
> IsIsomorphic(W1, W2);
true
> IsCoxeterIsomorphic(W1, W2);
false
```

An example of Coxeter isomorphic groups which are not Cartan equivalent:

```
> W1 := CoxeterGroup("B3");
> W2 := CoxeterGroup("C3");
> IsIsomorphic(W1, W2);
true
> IsCoxeterIsomorphic(W1, W2);
true [ 1, 2, 3 ]
> IsCartanEquivalent(W1, W2);
false
```

`RootSystem(W)`

The underlying root system of the permutation Coxeter group W.

`RootDatum(W)`

The root datum of the permutation Coxeter group W. If W does not have a root datum, an error is flagged.

Example H104E7

```
> W := CoxeterGroup("C5");
> RootSystem(W);
Root system of type C5
> RootDatum(W);
Root datum of type C5
>
> W := CoxeterGroup("H4");
> RootSystem(W);
Root system of type H4
>  RootDatum(W);
Error: This group does not have a root datum
```

`CartanName(W)`

The Cartan name of the finite or affine Coxeter group W (Section 101.6).

`CoxeterDiagram(W)`

Print the Coxeter diagram of the finite or affine Coxeter group W (Section 101.6).

`DynkinDiagram(W)`

Print the Dynkin diagram of the permutation Coxeter group W. If W is not crystallographic, an error is flagged.

Example H104E8

```
> W := CoxeterGroup("F4");
> CartanName(W);
F4
> DynkinDiagram(W);
F4    1 - 2 =>= 3 - 4
> CoxeterDiagram(W);
F4    1 - 2 === 3 - 4
```

`CoxeterMatrix(W)`

The Coxeter matrix of the Coxeter group W.

`CoxeterGraph(W)`

The Coxeter graph of the Coxeter group W.

`CartanMatrix(W)`

The Cartan matrix of the permutation Coxeter group W.

`DynkinDigraph(W)`

The Dynkin digraph of the permutation Coxeter group W.

`Rank(W)`

`NumberOfGenerators(W)`

The rank of the Coxeter group W.

`NumberOfPositiveRoots(W)`

`NumPosRoots(W)`

The number of positive roots of the Coxeter group W.

`Dimension(W)`

The dimension of the permutation Coxeter group W, ie. the dimension of the root space.

Example H104E9

```
> R := StandardRootSystem("A", 4);
> W := CoxeterGroup(R);
> Rank(W);
4
> Dimension(W);
5
```

`ConjugacyClasses(W)`

The conjugacy classes of the finite Coxeter group W. This uses the algorithm of [GP00].

`FundamentalGroup(W)`

The fundamental group of the permutation Coxeter group W. The roots and coroots of W must have integral components.

`IsogenyGroup(W)`

The isogeny group of the permutation Coxeter group W. The roots and coroots of W must have integral components.

`CoisogenyGroup(W)`

The coisogeny group of the permutation Coxeter group W. The roots and coroots of W must have integral components.

`BasicDegrees(W)`

The degrees of the basic invariant polynomials of the Coxeter group W. These are computed using the table in [Car72, page 155].

`BasicCodegrees(W)`

The basic codegrees of the Coxeter group W. These are computed using the algorithm in [LT09].

Example H104E10

The product of the basic degrees is the order of the Coxeter group; the sum of the basic degrees is the sum of the rank and the number of positive roots.

```
> W := CoxeterGroup("E6");
> degs := BasicDegrees(W);
> degs;
[ 2, 5, 6, 8, 9, 12 ]
> &*degs eq #W;
true
> &+degs eq NumPosRoots(W) + Rank(W);
true
```

`BruhatLessOrEqual(x, y)`

If Coxeter group element x is less than or equal to y in the Bruhat order [Deo77]. Suppose x is an element of the Coxeter group W. The Bruhat order is the partial order generated by the relations: $x \leq xw$ if $l(x) < l(xw)$, and $xw \leq x$ if $l(xw) < l(x)$, for $x \in W$ and w a reflection. If $l(xw) = l(x)+1$, then x is called a *Bruhat descendant* of xw. The algorithm used is a straightforward recursive procedure.

`BruhatDescendants(x)`

z	GrpPermElt	*Default :*

Let x be an element of the Coxeter group W, then the returned set S contains the Bruhat descendants of x. If $l(yw) = l(y) + 1$, then y is called a *Bruhat descendant* of yw. If the optional parameter z is set, only those descendants y with $z \leq y$ are returned. Algorithm: For each fundamental reflection in x it is tested whether leaving it out decreases the length of x by exactly 1. If so, it is included in the result. In particular, this algorithm does not use `BruhatLessOrEqual`.

`BruhatDescendants(X)`

z GrpPermElt *Default :*

Let X consist of elements of the Coxeter group W, then the returned set S contains the Bruhat descendants of every element of X.

If the optional parameter z is set, only those w are returned for which $z \leq w$ in the Bruhat ordering.

Example H104E11

Bruhat descendants:

```
> R := RootDatum("D4" : Isogeny := "SC");
> W := CoxeterGroup(GrpPermCox, R);
> Wfp,phi := CoxeterGroup(GrpFPCox, W);
> x := W.1*W.3*W.2*W.4*W.2*W.2*W.2*W.1;
> Eltseq(phi(x));
[ 1, 3, 2, 4, 2, 1 ]
> S := BruhatDescendants(x);
> { Eltseq(phi(w)) : w in S };
{
     [ 1, 3, 2, 4, 2 ],
     [ 3, 2, 4, 2, 1 ],
     [ 1, 2, 4, 2, 1 ],
     [ 1, 3, 2, 1, 4 ],
     [ 1, 3, 4, 2, 1 ]
}
```

104.5 Properties of Coxeter Groups

`IsFinite(W)`

Returns `true` if, and only if, the Coxeter group W is finite.

`IsAffine(W)`

Returns `true` if, and only if, the Coxeter group W is affine (Section 101.6).

`IsHyperbolic(W)`

Returns `true` if, and only if, the Coxeter group W is hyperbolic (Section 101.7).

`IsCompactHyperbolic(W)`

Returns `true` if, and only if, the Coxeter group W is compact hyperbolic (Section 101.7).

`IsIrreducible(W)`

Returns `true` if, and only if, the Coxeter group W is irreducible.

`IsSemisimple(W)`

Returns `true` if, and only if, the permutation Coxeter group W is semisimple, i.e. its rank is equal to its dimension.

`IsCrystallographic(W)`

Returns `true` if, and only if, the permutation Coxeter group W is crystallographic, i.e. if the corresponding reflection representation is defined over the integers.

`IsSimplyLaced(W)`

Returns `true` if, and only if, the Coxeter group W is simply laced, i.e. its Coxeter graph has no labels.

Example H104E12

```
> W := CoxeterGroup(GrpFPCox, HyperbolicCoxeterMatrix(22));
> IsFinite(W);
false
> IsAffine(W);
false
> IsHyperbolic(W);
true
> IsCompactHyperbolic(W);
false
> IsIrreducible(W);
true
> IsSimplyLaced(W);
true
> W := CoxeterGroup("A2 D4");
> IsIrreducible(W);
false
> IsSemisimple(W);
true
> IsCrystallographic(W);
true
> IsSimplyLaced(W);
true
```

104.6 Operations on Elements

See Chapter 75 for general functions for finitely presented groups or Chapter 63 for general functions for permutation groups.

Unlike groups of type `GrpFP`, elements of a group of type `GrpFPCox` are always converted into the normal form of Section 104.1.1.

Example H104E13

Arithmetic with words.

```
> W<[s]> := CoxeterGroup(GrpFPCox, "G2");
> w1 := W![2,1,2,1,2] ;
> w1;
s[2] * s[1] * s[2] * s[1] * s[2]
> w2 := W![1,2,2,1,2,1];
> w2;
s[2] * s[1]
> w1 * w2;
s[1] * s[2] * s[1]
> W![1,2,1,2,1,2] eq W![2,1,2,1,2,1];
true
```

`#w`

`Length(w)`

`CoxeterLength(w)`

The length of w as an element of the Coxeter group W, ie. the number of positive roots of W which become negative under the action of w. The # operator does not work for permutation Coxeter group elements.

`LongestElement(W)`

The unique longest element of the Coxeter group W.

`CoxeterElement(W)`

The Coxeter element of the Coxeter group W, ie. the product of the generators of W.

`CoxeterNumber(W)`

The Coxeter number of the irreducible Coxeter group W (see [Car93, page 20]).

Example H104E14

```
> W<[s]> := CoxeterGroup(GrpFPCox, "F4");
> LongestElement(W);
s[1] * s[2] * s[1] * s[3] * s[2] * s[1] * s[3] * s[2] * s[3] * s[4] * s[3] *
s[2] * s[1] * s[3] * s[2] * s[3] * s[4] * s[3] * s[2] * s[1] * s[3] * s[2] *
s[3] * s[4]
> CoxeterElement(W);
s[1] * s[2] * s[3] * s[4]
> W := CoxeterGroup("E8");
> Length(W, LongestElement(W));
120
> Length(W, CoxeterElement(W));
8
```

The Coxeter number can be described in a variety of ways.

```
> W := CoxeterGroup("D5");
> CoxeterNumber(W) eq Order(CoxeterElement(W));
true
> CoxeterNumber(W) eq #Roots(W) / Rank(W);
true
> R := RootDatum(W);
> CoxeterNumber(W) eq &+Eltseq(HighestRoot(R)) + 1;
true
```

`LeftDescentSet(W, w)`

The set of indices r of simple roots of the Coxeter group W such that the length of the product $s_r w$ is less than that of the element w.

`RightDescentSet(W, w)`

The set of indices r of simple roots of the Coxeter group W such that the length of the product ws_r is less than that of the element w.

Example H104E15

```
> W := CoxeterGroup("A5");
> x := W.1*W.2*W.4*W.5;
> LeftDescentSet(W, x);
{ 1, 4 }
> RightDescentSet(W, x);
{ 2, 5 }
```

104.7 Roots, Coroots and Reflections

The functions in this section give access to the underlying root system (or datum) of a permutation Coxeter group. These functions do not apply to finitely presented Coxeter groups

Roots are stored as an indexed set

$$\{@\ \alpha_1, \ldots, \alpha_N, \alpha_{N+1}, \ldots, \alpha_{2N}\ @\},$$

where $\alpha_1, \ldots, \alpha_N$ are the positive roots in an order compatible with height; and $\alpha_{N+1}, \ldots, \alpha_{2N}$ are the corresponding negative roots (i.e. $\alpha_{i+N} = -\alpha_i$). The simple roots are $\alpha_1, \ldots, \alpha_n$ where n is the rank.

Many of these functions have an optional argument `Basis` which may take one of the following values

1. `"Standard"`: the standard basis for the (co)root space. This is the default.
2. `"Root"`: the basis of simple (co)roots.
3. `"Weight"`: the basis of fundamental (co)weights (see Subsection 105.8.3 below).

104.7.1 Accessing Roots and Coroots

`RootSpace(W)`

`CorootSpace(W)`

The (co)root space of the Coxeter group W. This can be a vector space over a field of characteristic zero (Chapter 28), or an integer lattice in the crystallographic case (Chapter 30). The (co)reflection group of W acts on the (co)root space.

`SimpleRoots(W)`

`SimpleCoroots(W)`

The simple (co)roots of the Coxeter group W as the rows of a matrix.

Example H104E16

```
> W := CoxeterGroup("G2");
> RootSpace(W);
Full Vector space of degree 2 over Rational Field
> CorootSpace(W);
Full Vector space of degree 2 over Rational Field
> SimpleRoots(W);
[1 0]
[0 1]
> SimpleCoroots(W);
[ 2 -3]
[-1  2]
> CartanMatrix(W);
[ 2 -1]
[-3  2]
```

`NumberOfPositiveRoots(W)`

`NumPosRoots(W)`

The number of positive roots of the Coxeter group W.

`Roots(W)`

`Coroots(W)`

`Basis` MonStgElt *Default :* "*Standard*"

An indexed set containing the (co)roots of the Coxeter group W.

`PositiveRoots(W)`

`PositiveCoroots(W)`

`Basis` MonStgElt *Default :* "*Standard*"

An indexed set containing the positive (co)roots of the Coxeter group W.

`Root(W, r)`

`Coroot(W, r)`

`Basis` MonStgElt *Default :* "*Standard*"

The rth (co)root of the Coxeter group W.

`RootPosition(W, v)`

`CorootPosition(W, v)`

`Basis` MonStgElt *Default :* "*Standard*"

If v is a (co)root of the Coxeter group W, this returns its position; otherwise it returns 0. These functions will try to coerce v, which can be a vector or a sequence representing a vector, into the appropriate vector space; v should be written with respect to the basis specified by the parameter `Basis`.

Example H104E17

```
> A := Matrix(2,3, [1,-1,0, -1,1,-1]);
> B := Matrix(2,3, [1,-1,1, 0,1,-1]);
> W := CoxeterGroup(A, B);
> Roots(W);
{@
    (1 -1  0),
    (-1  1 -1),
    (0  0 -1),
    (1 -1 -1),
    (2 -2 -1),
    (1 -1 -2),
    (-1  1  0),
    (1 -1  1),
    (0 0 1),
```

```
    (-1  1  1),
    (-2  2  1),
    (-1  1  2)
@}
> PositiveCoroots(W);
{@
    (1 -1  1),
    (0  1 -1),
    (1  2 -2),
    (2  1 -1),
    (1 0 0),
    (1  1 -1)
@}
> #Roots(W) eq 2*NumPosRoots(W);
true
> Root(W, 4);
(1 -1 -1)
> Root(W, 4 : Basis := "Root");
(2 1)
> RootPosition(W, [1,-1,-1]);
4
> RootPosition(W, [2,1] : Basis := "Root");
4
```

`HighestRoot(W)`

`HighestLongRoot(W)`

Basis MonStgElt *Default :* "*Standard*"

The unique (long) root of greatest height of the irreducible Coxeter group W.

`HighestShortRoot(W)`

Basis MonStgElt *Default :* "*Standard*"

The unique short root of greatest height of the irreducible Coxeter group W.

Example H104E18

```
> W := RootDatum("G2");
> HighestRoot(W);
(3 2)
> HighestLongRoot(W);
(3 2)
> HighestShortRoot(W);
(2 1)
```

`CoxeterForm(W)`

`DualCoxeterForm(W)`

`Basis` MonStgElt *Default : "Standard"*

The matrix of an inner product on the (co)root space of the finite Coxeter group W which is invariant under the action of W. This inner product is uniquely determined up to a constant on each irreducible component of W. The inner product is normalised so that the short roots in each crystallographic component have length one.

`AdditiveOrder(W)`

An additive order on the positive roots of the finite Coxeter group W, i.e. a sequence containing the numbers $1, \ldots, N$ in some order such that $\alpha_r + \alpha_s = \alpha_t$ implies t is between r and s. This is computed using the techniques of [Pap94].

104.7.2 Operations and Properties for Root and Coroot Indices

`Sum(W, r, s)`

The index of the sum of the rth and sth roots in the Coxeter group W, or 0 if the sum is not a root. In other words, if $t =$ `Sum(W,r,s)` $\neq 0$ then $\alpha_t = \alpha_r + \alpha_s$. The condition $\alpha_r \neq \pm\alpha_s$ must be satisfied. If W is noncrystallographic, an error is flagged.

`IsPositive(W, r)`

Returns `true` if, and only if, the rth (co)root of the Coxeter group W is a positive root.

`IsNegative(W, r)`

Returns `true` if, and only if, the rth (co)root of the Coxeter group W is a negative root.

`Negative(W, r)`

The index of the negative of the rth (co)root of the Coxeter group W. In other words, if $s =$ `Negative(W,r)` then $\alpha_s = -\alpha_r$.

`LeftString(W, r, s)`

Root indices in the Coxeter group W of the left string through α_s in the direction of α_r, i.e. the indices of $\alpha_s - \alpha_r, \alpha_s - 2\alpha_r, \ldots, \alpha_s - p\alpha_r$. In other words, this returns the sequence $[r_1, \ldots, r_p]$ where $\alpha_{r_i} = \alpha_s - i\alpha_r$ and $\alpha_s - (p+1)\alpha_r$ is not a root. The condition $\alpha_r \neq \pm\alpha_s$ must be satisfied. If W is noncrystallographic, an error is flagged.

`RightString(W, r, s)`

Root indices of the Coxeter group W of the left string through α_s in the direction of α_r, i.e. the indices of $\alpha_s + \alpha_r, \alpha_s + 2\alpha_r, \ldots, \alpha_s + q\alpha_r$. In other words, this returns the sequence $[r_1, \ldots, r_q]$ where $\alpha_{r_i} = \alpha_s + i\alpha_r$ and $\alpha_s + (q+1)\alpha_r$ is not a root. The condition $\alpha_r \neq \pm\alpha_s$ must be satisfied. If W is noncrystallographic, an error is flagged.

`LeftStringLength(W, r, s)`

The largest p such that $\alpha_s - p\alpha_r$ is a root of the Coxeter group W. The condition $\alpha_r \neq \pm\alpha_s$ must be satisfied. If W is noncrystallographic, an error is flagged.

`RightStringLength(W, r, s)`

The largest q such that $\alpha_s + q\alpha_r$ is a root of the Coxeter group W. The condition $\alpha_r \neq \pm\alpha_s$ must be satisfied. If W is noncrystallographic, an error is flagged.

Example H104E19

```
> W := RootDatum("G2");
> Sum(W, 1, Negative(W,5));
10
> IsPositive(W, 10);
false
> Negative(W, 10);
4
> P := PositiveRoots(W);
> P[1] - P[5] eq -P[4];
true
```

`RootHeight(W, r)`

`CorootHeight(W, r)`

The height of the rth (co)root of the Coxeter group W, i.e. the sum of the coefficients of α_r (respectively, $\alpha_r^\star$) with respect to the simple (co)roots.

`RootNorms(W)`

`CorootNorms(W)`

The sequence of squares of the lengths of the (co)roots of the Coxeter group W.

`RootNorm(W, r)`

`CorootNorm(W, r)`

The square of the length of the rth (co)root of the Coxeter group W.

`IsLongRoot(W, r)`

Returns `true` if, and only if, the rth root of the Coxeter group W is long, i.e. the rth coroot is short. An error is flagged unless W is irreducible and crystallographic.

`IsShortRoot(W, r)`

Returns `true` if, and only if, the rth root of the Coxeter group W is short, i.e. the rth coroot is long. An error is flagged unless W is irreducible and crystallographic.

Example H104E20

```
> W := RootDatum("G2");
> RootHeight(W, 5);
4
> F := CoxeterForm(W);
> v := VectorSpace(Rationals(),2) ! Root(W, 5);
> (v*F, v) eq RootNorm(W, 5);
true
> IsLongRoot(W, 5);
true
> LeftString(W, 1, 5);
[ 4, 3, 2 ]
> roots := Roots(W);
> for i in [1..3] do
>   RootPosition(W, roots[5]-i*roots[1]);
> end for;
4
3
2
```

104.7.3 Weights

`WeightLattice(W)`

`CoweightLattice(W)`

The (co)weight lattice of the Coxeter group W. The roots and coroots of W must have integral components.

`FundamentalWeights(W)`

`FundamentalCoweights(W)`

`Basis` MonStgElt *Default :* "*Standard*"

The fundamental (co)weights of the Coxeter group W. The roots and coroots of W must have integral components.

`IsDominant(R, v)`

`Basis` MonStgElt *Default : "Standard"*

Returns `true` if, and only if, v is a dominant weight for the root datum R, ie, a nonnegative integral linear combination of the fundamental weights.

`DominantWeight(W, v)`

`Basis` MonStgElt *Default : "Standard"*

The unique element in the W-orbit of the weight v which lies in the fundamental Weyl chamber, and the word in the generators which sends v to this element. The Coxeter group W must have a root datum. The weight v can be given either as a vector or as a sequence representing the vector and is coerced into the weight lattice first.

`WeightOrbit(W, v)`

`Basis` MonStgElt *Default : "Standard"*

The orbit of the weight v under the action of W. The Coxeter group W must have a root datum. The weight v can be given either as a vector or as a sequence representing the vector and is coerced into the weight lattice first.

Example H104E21

```
> W := CoxeterGroup("B3");
> DominantWeight(W, [1,-1,0] : Basis:="Weight");
(1 0 0)
[ 2, 3, 2, 1 ]
> #WeightOrbit(W, [1,-1,0] : Basis:="Weight");
6
```

104.8 Reflections

An element of a Coxeter group is called a *reflection* if it is conjugate to one of the Coxeter generators.

In a permutation Coxeter group, the root α acts on the root space via the reflection s_α; the coroot $\alpha^\star$ acts on the coroot space via the coreflection $s_\alpha^\star$.

`IsReflection(w)`

Returns `true` if, and only if, w is a reflection, i.e. w is conjugate to a Coxeter generator. If w is in a permutation Coxeter group, the root, coroot and root index are also returned.

`Reflections(W)`

The sequence of reflections in the finite Coxeter group W. If W is a permutation Coxeter group, the rth reflection in the sequence corresponds to the rth (co)root.

Example H104E22

```
> W<a,b> := CoxeterGroup(GrpFPCox, "A2");
> Reflections(W);
[ a, b, a * b * a, a, b, a * b * a ]
> IsReflection(a*b);
false
```

`SimpleReflections(W)`

The sequence of simple reflections in the Coxeter group W, ie, the generators of W.

`SimpleReflectionPermutations(W)`

The sequence of simple reflections in the permutation Coxeter group W, ie, the generators of W.

`Reflection(W, r)`

`ReflectionPermutation(W, r)`

The reflection in permutation Coxeter group W corresponding to the rth (co)root. If $r = 1, \ldots, n$, this is a generator of W.

`SimpleReflectionMatrices(W)`

`SimpleCoreflectionMatrices(W)`

`Basis` MonStgElt *Default :* "*Standard*"

The matrices giving the action of the simple (co)roots on the (co)root space of the permutation Coxeter group W.

`ReflectionMatrices(W)`

`CoreflectionMatrices(W)`

`Basis` MonStgElt *Default :* "*Standard*"

The matrices giving the action of the (co)roots on the (co)root space of the permutation Coxeter group W.

`ReflectionMatrix(W, r)`

`CoreflectionMatrix(W, r)`

`Basis` MonStgElt *Default :* "*Standard*"

The matrix giving the action of the rth (co)root on the (co)root space of the permutation Coxeter group W.

`ReflectionWords(W)`

The sequence of words in the simple reflections for all the reflections of the Coxeter group W. These words are given as sequences of integers. In other words, if $a = [a_1, \ldots, a_l] =$ `ReflectionWords(W)[r]`, then $s_{\alpha_r} = s_{\alpha_{a_1}} \cdots s_{\alpha_{a_l}}$.

`ReflectionWord(W, r)`

The word in the simple reflections for the rth reflection of the Coxeter group W. The word is given as a sequence of integers. In other words, if $a = [a_1, \ldots, a_l] =$ `ReflectionWord(W,r)`, then $s_{\alpha_r} = s_{\alpha_{a_1}} \cdots s_{\alpha_{a_l}}$.

Example H104E23

```
> W := CoxeterGroup("B3");
> IsReflection(W.1*W.2);
false
> mx := ReflectionMatrix(W, 4);
> perm := Reflection(W, 4);
> wd := ReflectionWord(W, 4);
> rt := VectorSpace(Rationals(), 3) ! Root(W,2);
> RootPosition(W, rt * mx) eq 2^perm;
true
> perm eq &*[ Reflection(W, r) : r in wd ];
true
>
> mx := CoreflectionMatrix(W, 4);
> CorootPosition(W, Coroot(W,2) * mx) eq 2^perm;
true
```

104.9 Reflection Subgroups

A *reflection subgroup* of a Coxeter group is a subgroup which is generated by a set of reflections. Note that reflection subgroups are also Coxeter groups. The most important class of reflection subgroups are the *standard parabolic subgroups*, which are generated by a subset of the simple roots. Given a set of indices $J \subseteq \{1, \ldots,$ `Rank(W)`$\}$, the corresponding standard parabolic is denoted W_J. A *parabolic subgroup* is a subgroup which is conjugate to a standard parabolic subgroup. Note that in a reflection subgroup, the elements are given as permutations of the roots of the *larger* group.

Most of the functions in this section are currently only implemented for permutation Coxeter groups with a root *datum* (rather than a root system).

`ReflectionSubgroup(W, a)`

The reflection subgroup of the permutation Coxeter group W generated by the roots $\alpha_{a_1}, \ldots, \alpha_{a_k}$ where $a = \{a_1, \ldots, a_k\}$ is a set of integers. This only works if W has an underlying root datum.

`ReflectionSubgroup(W, s)`

The reflection subgroup of the permutation Coxeter group W generated by simple roots $\alpha_{s_1}, \ldots, \alpha_{s_k}$ where $s = [s_1, \ldots, s_k]$ is a *sequence* of integers. In this version the roots must be simple in the root subdatum (ie. none of them may be a summand of another) otherwise an error is signalled. The simple roots will appear in the reflection subgroup in the given order. This only works if W has an underlying root datum.

`StandardParabolicSubgroup(W, J)`

The standard parabolic subgroup of the Coxeter group W generated by the simple roots $\alpha_{j_1}, \ldots, \alpha_{j_k}$ where $J = \{j_1, \ldots, j_k\} \subseteq \{1, \ldots, \texttt{Rank(W)}\}$. This function works for both finitely presented and permutation Coxeter groups.

`IsReflectionSubgroup(W, H)`

Returns `true` if, and only if, H is a reflection subgroup of the permutation Coxeter group W.

`IsParabolicSubgroup(W, H)`

Returns `true` if, and only if, H is a parabolic subgroup of the permutation Coxeter group W.

`IsStandardParabolicSubgroup(W, H)`

Returns `true` if, and only if, H is a standard parabolic subgroup of the permutation Coxeter group W.

`Overgroup(H)`

The overgroup of H, ie. the Coxeter group whose roots are permuted by the elements of the permutation Coxeter subgroup H.

`Overdatum(H)`

The root datum whose roots are permuted by the elements of the permutation Coxeter subgroup H.

`LocalCoxeterGroup(H)`

Given a Coxeter subgroup H this returns the Coxeter group L isomorphic to H but acting on the roots of H itself rather than the roots of its overgroup, together with the isomorphism $L \to H$.

Example H104E24

```
> W := CoxeterGroup("A4");
> P := StandardParabolicSubgroup(W, {1,2});
> Overgroup(P) eq W;
true
> L, h := LocalCoxeterGroup(P);
> hinv := Inverse(h);
> L.1;
(1, 4)(2, 3)(5, 6)
> h(L.1);
(1, 11)(2, 5)(6, 8)(9, 10)(12, 15)(16, 18)(19, 20)
> hinv(h(L.1));
(1, 4)(2, 3)(5, 6)
```

`Transversal(W, H)`

The indexed set of (right) coset representatives of the reflection subgroup H of the Coxeter group W. This contains the unique element of shortest length in each coset. The algorithm is due to Don Taylor (personal communication).

`TransversalWords(W, H)`

The indexed set of words of (right) coset representatives of the reflection subgroup H of the Coxeter group W. The algorithm is due to Don Taylor (personal communication).

`TransversalElt(W, H, x)`

The representative of the coset Hx in the Coxeter group W. This is the unique element of Hx of shortest length in W and also the unique element of Hx which sends every positive root of H to another positive root. The algorithm is due to Don Taylor (personal communication).

Example H104E25

```
> W := CoxeterGroup("A4");
> P := StandardParabolicSubgroup(W, {1,2});
> x := W.1 * W.2 * W.3;
> x := TransversalElt(W, P, x);
> x eq W.3;
true
> x in Transversal(W, P);
true
```

`TransversalElt(W, x, H)`

The representative of the coset xH in the Coxeter group W. This is the unique element of xH of shortest length in W and also the unique element of xH which sends every positive root of H to another positive root.

`TransversalElt(W, H, x, J)`

The representative of the coset HxJ in the Coxeter group W. This is the unique element of HxJ of shortest length in W and also the unique element of HxJ which sends every positive root of HJ to another positive root.

`Transversal(W, J)`

`Transversal(W, J, L)`

The set of right coset representatives of minimal length for the standard parabolic subgroup $W_J \leq W$. In the first form W must be finite and the result is a full transversal. In the second form W may be infinite, but the transversal produced is limited to words of length at most L.

`Transversal(W, J, K)`

The sequence of W_J, W_K-double cosets representatives of minimal length in W. Restricted to W finite. The second return value gives the generators of the standard parabolic subgroup $W_J \cap W_K^d$ for each double coset representative d.

`DirectProduct(W1, W2)`

The direct product of the Coxeter groups W_1 and W_2.

`Dual(W)`

The dual of the Coxeter group W, obtained by swapping the roots and coroots.

Example H104E26

```
> W1 := CoxeterGroup("G2");
> W2 := CoxeterGroup("C3");
> DirectProduct(W1, Dual(W2));
Coxeter group: Permutation group acting on a set of cardinality 30
Order = 576 = 2^6 * 3^2
    (1, 7)(2, 5)(3, 4)(8, 11)(9, 10)
    (1, 3)(2, 8)(5, 6)(7, 9)(11, 12)
    (13, 22)(14, 16)(17, 20)(19, 21)(23, 25)(26, 29)(28, 30)
    (13, 16)(14, 23)(15, 17)(18, 21)(22, 25)(24, 26)(27, 30)
    (14, 19)(15, 24)(16, 21)(23, 28)(25, 30)
> W1 := CoxeterGroup(GrpFPCox, "G2");
> W2 := CoxeterGroup(GrpFPCox, "A2");
> DirectProduct(W1, W2);
Coxeter group: Finitely presented group on 4 generators
Relations
```

```
$.1 * $.2 * $.1 = $.2 * $.1 * $.2
$.1 * $.3 = $.3 * $.1
$.1 * $.4 = $.4 * $.1
$.2 * $.3 = $.3 * $.2
$.2 * $.4 = $.4 * $.2
$.3 * $.4 * $.3 = $.4 * $.3 * $.4
$.1^2 = Id($)
$.2^2 = Id($)
$.3^2 = Id($)
$.4^2 = Id($)
```

104.10 Root Actions

The functions in this section give access to the action on the underlying root system (or datum) of a permutation Coxeter group. These functions do not apply to finitely presented Coxeter groups

In the following functions, the optional parameter `Basis` determines which basis the roots are given with respect to: `"Standard"` for the standard basis of the root space; `"Root"` for the basis of simple (co)roots; `"Weight"` for the basis of simple (co)weights.

`RootGSet(W)`
`CorootGSet(W)`

`Basis` MonStgElt *Default :* "*Standard*"

The G-set of the Coxeter group W acting on the (co)roots.

Example H104E27

```
> W := CoxeterGroup("B3");
> X := RootGSet(W);
> r := Root(W, 5);
> r;
(0 1 1)
> Image(W.1, X, r);
(1 1 1)
```

`RootAction(W)`
`CorootAction(W)`

`Basis` MonStgElt *Default :* "*Standard*"

The map $X \times W \to X$ giving the action of the Coxeter group W on the (co)root space X.

Example H104E28

```
> W := CoxeterGroup("B3");
> act := CorootAction(W);
> act([1,-2,1], W.1);
(-1 -1  1)
```

`ReflectionGroup(W)`

`CoreflectionGroup(W)`

`Basis` MonStgElt *Default :* "*Standard*"

The Coxeter group W as a real reflection group (ie. as a matrix group over some subfield of $\mathbf{R}$) acting on the (co)root space, and the isomorphism from W to the (co)reflection group.

Example H104E29

```
> W := CoxeterGroup("B3");
> _, h := ReflectionGroup(W);
> W.1*W.3;
(1, 10)(2, 8)(3, 12)(4, 7)(5, 6)(11, 17)(13, 16)(14, 15)
> h(W.1*W.3);
[-1  0  0]
[ 1  1  2]
[ 0  0 -1]
```

104.11 Standard Action

Every finite Coxeter group W has a standard action. For example, the standard action group of a Coxeter group of type A_n is the symmetric group of degree $n+1$ acting on $\{1, \ldots, n\}$.

`StandardAction(W)`

The standard action of the finite Coxeter group W.

`StandardActionGroup(W)`

The group G of the standard action of the finite Coxeter group W, together with an isomorphism $W \to G$.

Example H104E30

```
> W := CoxeterGroup("A3");
> G, h := StandardActionGroup(W);
> IsSymmetric(G);
true
> h(W.1); h(W.2); h(W.3);
(1, 2)
(2, 3)
(3, 4)
```

104.12 Braid Groups

`BraidGroup(W)`

The braid group B of the Coxeter group W as a finitely presented group, together with the natural map $W \to B$. Words in the braid group are not automatically normalised. However, the braid group of type A_n with normalisation can be constructed with the command `BraidGroup(n+1)` (see Chapter 78).

`PureBraidGroup(W)`

Returns the pure braid group of the Coxeter group W, ie. the kernel of the epimorphism from the braid group of W to W. Words in the pure braid group are not automatically normalised.

Example H104E31

```
> W<a,b,c> := CoxeterGroup(GrpFPCox, "B3");
> W;
Coxeter group: Finitely presented group on 3 generators
Relations
    a * b * a = b * a * b
    a * c = c * a
    (b * c)^2 = (c * b)^2
    a^2 = Id($)
    b^2 = Id($)
    c^2 = Id($)
> B<x,y,z> := BraidGroup(W);
> B;
Finitely presented group B on 3 generators
Relations
    x * y * x = y * x * y
    x * z = z * x
    (y * z)^2 = (z * y)^2
> P := PureBraidGroup(W);
```

```
> P;
Finitely presented group P on 3 generators
Generators as words in group B
    P.1 = x^2
    P.2 = y^2
    P.3 = z^2
```

104.13 W-graphs

Given a Coxeter system (W, S), a W-graph is a (directed or undirected) graph with vertex labels and edge weights. The label attached to a vertex v is a subset of S (called the *descent set* of v) and the edge weights are scalars (usually integers).

A W-graph must determine a representation of the Hecke algebra $H = H\langle q\rangle$ of the associated Coxeter system. The vertices of the W-graph can be identified with basis elements of the representation space, and by the conventions adopted here the action of the generator T_s of H associated with an element $s \in S$ on a basis element v is given by

$$v * T_s = \begin{cases} (-q^{-1}) * v & \text{if } s \text{ is in the descent set of } v, \\ q * v + \sum'(m * u) & \text{if } s \text{ is not in the descent set of } v, \end{cases}$$

where $\sum'$ indicates the sum over all edges with terminal vertex equal to v for which s is in the descent set of the initial vertex u, and m is the weight of the edge.

For the Coxeter group calculations involved in these functions we need to know how the generators $s \in S$ act on the set of elementary roots (see [Bri98]).

MAGMA has a function `ReflectionTable` that provides the necessary information. Specifically, let W be a finitely presented Coxeter group with N elementary roots (numbered from 1 to N) and r simple reflections (numbered 1 to r). If we define

```
eltroots:=ReflectionTable(W);
```

then for $i \in \{1, \ldots, r\}$ and $j \in \{1, \ldots, N\}$, `eltroots[i,j] = k` if the i-th simple reflection takes the j-th elementary root to the k-th elementary root, or to a non-elementary root if $k = 0$, or to a negative root if $k < 0$. (This last alternative occurs if and only if $j = i$ and $k = -i$.) Knowing the table `eltroots` makes it quick and easy to do symbolic computation with elements of W, represented as sequences of integers in $\{1, \ldots, r\}$ (corresponding to words in S).

`SetVerbose("WGraph", v)`

Set the verbose printing to level v for all W-graph related functions. A level of 2 means that informative messages and progress information will be printed durng a computation.

Sometimes it is convenient to use 'mij-sequences' to specify Coxeter groups. The mij-sequence consists of the on or below diagonal entries in the Coxeter matrix. Thus if `seq` is the mij-sequence and M the Coxeter matrix then

```
M := SymmetricMatrix(seq);
```
and
```
seq := &cat[[M[i,j] :  j in [1..i]] :  i in [1..Rank(W)]];
```

`Mij2EltRootTable(seq)`

Return the elementary root action table for the Coxeter group defined by the given mij-sequence.

`Name2Mij(name)`

The mij-sequence of the Coxeter groups of type name.

Example H104E32

```
> e6:=[1,3,1,2,3,1,2,3,2,1,2,2,2,3,1,2,2,3,2,2,1];
> E6 := CoxeterGroup(GrpFPCox, SymmetricMatrix(e6) );
> ReflectionTable(E6) eq Mij2EltRootTable(e6);
true
```

The functions defined in this section are mainly concerned with W-graph posets. The motivating example for this concept is the set of all standard tableaux corresponding to a given partition, the partial order being dominance. By definition, if P is a W-graph poset then P must be in one-to-one correspondence with a basis for an H-module V (where H is the Hecke algebra associated with the given Coxeter system). In the standard tableaux example, this module is the Specht module; hence in the general case we refer to the module V as GSM(P) (for generalized Specht module). For each $v \in P$ the set S must be the disjoint union of two sets $A(v)$ and $D(v)$, the ascents and descents of v. There must be a function $(s, v) \mapsto sv$ from $S \times P$ to P such that the action of H on GSM(P) satisfies the following rules (for all $s \in S$ and $v \in P$):

$$v * T_s = \begin{cases} sv & \text{if } sv > v, \\ sv + (q - q^{-1}) * v & \text{if } sv < v, \\ -q^{-1} * v & \text{if } sv = v \text{ and } s \in D(v), \\ q * v + q * \langle \text{earlier} \rangle & \text{if } sv = v \text{ and } s \in A(v), \end{cases}$$

where $\langle$earlier$\rangle$ denotes a linear combination of $\{u \in P \mid u < v\}$ with coefficients that are polynomials in q. For each $s \in A(v)$ either $sv = v$ or $sv > v$, and for each $s \in D(v)$ either $sv < v$ or $sv = v$. This (admittedly strange) definition is motivated by the fact that Specht modules satisfy it. If v is a standard tableau corresponding to a partition of n then a number i in $\{1, \ldots, n-1\}$ is an ascent of v if $i+1$ is in a later column of t than i, and is a descent of v if $i+1$ is in a lower row of t than i. The fact that Specht modules satisfy the formulas above is proved in the literature (e.g. Mathas' book), except that in the "weak ascent" case ($sv = v$ and $s \in A(v)$) it is not proved that the polynomial coefficients of $\{u \in P \mid u < v\}$ are all divisible by q. The fact that they are is a theorem of V. M. Nguyen (PhD thesis, University of Sydney, 2010). It turns out that there is an algorithm by

which a W-graph may be constructed from a W-graph poset, the W-graph being uniquely determined by the function $(s, v) \mapsto sv$ from $S \times P \to P$ and the descent/ascent sets. The polynomial coefficients in the weak ascent case are not required. Of course the H-module determined by the resulting W-graph is isomorphic to GSM(P).

`Partition2WGtable(pi)`

Returns the W-graph table and the Weyl group for the partition `pi`, where `pi` is a nonincreasing sequence $[a_1, a_2, \ldots, a_k]$ of positive integers. It returns the table corresponding to the W-graph poset of standard tableaux of the given shape and the finitely presented Coxeter group of type A_n, where $n + 1 = \sum a_i$.

`WGtable2WG(table)`

Convert a W-graph table to a W-graph.

`TestWG(W,wg)`

`TestWG(tp,wg)`

This procedure can be used to test whether a presumed undirected or directed W-graph is indeed a W-graph, where W is a finitely presented Coxeter group of type `tp`. Two input values are required: the Coxeter group W (or its type) and the W-graph. When applied to the W-graph produced by the `WGtable2WG` function, this tests whether the input table did genuinely correspond to a W-graph poset.

For example,

Example H104E33

```
> SetVerbose("WGraph",2);
> wtable, W :=Partition2WGtable([4,4,3,1]);
> wg := WGtable2WG(wtable);
> TestWG(W,wg);
```

which should cause the word `true` to be printed 66 times (as the defining relations of the Hecke algebra are checked).

Given a Coxeter system (W, S) and an element $w \in W$, let P be the set $\{x \in W \mid \text{length}(wx^{-1}) = \text{length}(w) - \text{length}(x)\}$, considered as a poset under the Bruhat order on W. Given also a subset J of $\{t \in S \mid \text{length}(wt) > \text{length}(w)\}$, for each $x \in P$ we define $D(x)$ to be union of $\{s \in S \mid \text{length}(sx) < \text{length}(x)\}$ and $\{s \in S \mid sx = xt \text{ for some } t \in J\}$. If P is now a W-graph poset with the sets $D(x)$ as the descent sets then we say that w is a W-graph determining element relative to J.

For example, suppose that (W, S) is of type A_n, and given a partition of $n + 1$ let t be the (unique) standard tableau whose column group is generated by a subset of S. Let w be the maximal length element such that the tableau wt is standard. Then w is a W-graph determining element with respect to the set J consisting of those $s \in S$ that are in the column stabilizer of t.

Other examples (for any Coxeter system with finite W) are provided by the distinguished left coset representatives of maximal length for standard parabolic subgroups W_K (where the set J may be taken to be either K or the empty set).

`WGelement2WGtable(g,K)`

Returns the W-graph table and W-graph ideal of a W-graph determining element g, subset K.

Example H104E34

```
> b5 := [1,4,1,2,3,1,2,2,3,1,2,2,2,3,1];
> b5mat := SymmetricMatrix(b5);
> W := CoxeterGroup(GrpFPCox, b5mat );
> table, _ := WGelement2WGtable(W![5,4,3,2,1,2,3,4,5],{});
> wg := WGtable2WG(table);
> TestWG(W,wg);
true <1, 2> 4
true <2, 3> 3
true <3, 4> 3
true <4, 5> 3
```

`GetCells(wg)`

Return the cells of the W-graph.

`InduceWG(W,wg,seq)`

Induce a W-graph from a standard parabolic subgroup.

`InduceWGtable(J, table, W)`

Returns the table of the W-graph induced from the table of a parabolic subgroup defined by J.

`IsWGsymmetric(dwg)`

Test a W-graph for symmetry. If the graph is symmetric the second return value is the undirected version of the W-graph.

`MakeDirected(uwg)`

Convert an undirected W-graph to a directed W-graph.

`TestHeckeRep(W,r)`

Tests whether the matrices in r satisfy the defining relations of the Hecke algebra of the Coxeter group W.

`WG2GroupRep(wg)`

The matrix representation of a W-graph.

`WG2HeckeRep(W,wg)`

Returns a sequence of sparse matrices that satisfy the defining relations of the Hecke algebra.

`WGidealgens2WGtable(dgens,K)`

Returns the W-graph table and W-graph ideal of a W-graph determining generators dgens and subset K.

Example H104E35

In type E_6 we start with a rank 3 standard parabolic subgroup. The set of minimal coset representatives is a (single-generator) W-graph ideal, corresponding to the representation induced from the trivial representation of the parabolic. We compute the W-graph and find the cells. The bottom cell is necessarily an ideal in the weak order. It turns out that 3 elements are required to generate it; we can use them to test the function `WGidealgens2WGtable`.

```
> mij:=[1,3,1,2,3,1,2,3,2,1,2,2,2,3,1,2,2,3,2,2,1];
> E6 := CoxeterGroup(GrpFPCox, SymmetricMatrix(mij) );
> J := {1,3,5};
> drs := Transversal(E6,J);
> ttt := WGidealgens2WGtable([drs[1398],drs[156],drs[99]],J);
> nwg := WGtable2WG(ttt);
> TestWG(E6,nwg);
true <1, 2> 3
true <2, 3> 3
true <2, 4> 3
true <4, 5> 3
true <3, 6> 3
```

`WriteWG(file,uwg)`

`WriteWG(file,dwg)`

Writes the W-graph to a file.

104.14 Related Structures

In this section functions for creating other structures from a permutation Coxeter group are briefly listed. See the appropriate chapters of the Handbook for more details.

`CoxeterGroup(GrpFP, W)`

`Presentation(W)`

The finitely presented group isomorphic to the permutation Coxeter group W. See Chapter 75.

`ReflectionGroup(W)`

`CoxeterGroup(GrpMat, W)`

The reflection group isomorphic to the Coxeter group W. See Chapter 105.

`LieAlgebra(W, R)`

The reductive Lie algebra over the ring R with Weyl group W. If W is noncrystallographic, an error is flagged. See Section 106.5.1.

`GroupOfLieType(W, R)`

The group of Lie type over the ring R with Weyl group W. The roots and coroots of W must have integral components. See Chapter 109.

104.15 Bibliography

[**BH93**] Brigitte Brink and Robert B. Howlett. A finiteness property and an automatic structure for Coxeter groups. *Math. Ann.*, 296(1):179–190, 1993.

[**Bou68**] N. Bourbaki. *Éléments de mathématique. Fasc. XXXIV. Groupes et algèbres de Lie. Chapitre IV: Groupes de Coxeter et systèmes de Tits. Chapitre V: Groupes engendrés par des réflexions. Chapitre VI: Systèmes de racines.* Hermann, Paris, 1968.

[**Bri98**] Brigitte Brink. The set of dominance-minimal roots. *J. Algebra*, 206(2):371–412, 1998.

[**Car72**] Roger W. Carter. *Simple groups of Lie type.* John Wiley & Sons, London-New York-Sydney, 1972. Pure and Applied Mathematics, Vol. 28.

[**Car93**] Roger W. Carter. *Finite groups of Lie type.* John Wiley & Sons, Chichester, 1993. Conjugacy classes and complex characters, Reprint of the 1985 original, A Wiley-Interscience Publication.

[**Deo77**] V.V. Deodhar. Some Characteristics of Bruhat Ordering on a Coxeter group and determination of the Relative Möbius Function. *Inventiones Math.*, 39:179–198, 1977.

[**GHL+96**] Meinolf Geck, Gerhard Hiss, Frank Lübeck, Gunter Malle, and Götz Pfeiffer. CHEVIE—a system for computing and processing generic character tables. *Appl. Algebra Engrg. Comm. Comput.*, 7(3):175–210, 1996. Computational methods in Lie theory (Essen, 1994).

[**GP00**] Meinolf Geck and Götz Pfeiffer. *Characters of finite Coxeter groups and Iwahori-Hecke algebras*, volume 21 of *London Mathematical Society Monographs. New Series*. The Clarendon Press Oxford University Press, New York, 2000.

[**LT09**] G. I. Lehrer and D. E. Taylor. *Unitary Reflection Groups*, volume 20 of *Australian Mathematical Society Lecture Series*. Cambridge University Press, Cambridge, 2009.

[**Pap94**] Paolo Papi. A characterization of a special ordering in a root system. *Proc. Amer. Math. Soc.*, 120(3):661–665, 1994.

105 REFLECTION GROUPS

105.1 Introduction 3179

105.2 Construction of Pseudo-reflections 3179

PseudoReflection(a, b) 3180
Transvection(a, b) 3180
Reflection(a, b) 3180
IsPseudoReflection(r) 3180
IsTransvection(r) 3180
IsReflection(r) 3180
IsReflectionGroup(G) 3180

105.2.1 Pseudo-reflections Preserving Reflexive Forms 3182

SymplecticTransvection(a, alpha) 3182
UnitaryTransvection(a, alpha) 3182
UnitaryReflection(a, zeta) 3183
OrthogonalReflection(a) 3183

105.3 Construction of Reflection Groups 3184

PseudoReflectionGroup(A, B) 3184

105.4 Construction of Real Reflection Groups 3185

ReflectionGroup(M) 3185
ReflectionGroup(G) 3185
ReflectionGroup(C) 3185
ReflectionGroup(D) 3185
ReflectionGroup(N) 3185
IrreducibleReflectionGroup(X, n) 3185
ReflectionGroup(R) 3186
ReflectionGroup(W) 3187
ReflectionGroup(W) 3187

105.5 Construction of Finite Complex Reflection Groups . . . 3188

ShephardTodd(n) 3189
ComplexReflectionGroup(C) 3190
ComplexReflectionGroup(X, n) 3190
ShephardTodd(m, p, n) 3192
ImprimitiveReflectionGroup(m, p, n) 3192
ComplexRootMatrices(k) 3193
ComplexRootMatrices(m, p, n) 3193
ComplexCartanMatrix(k) 3194
ComplexCartanMatrix(m, p, n) 3194
BasicRootMatrices(C) 3194
CohenCoxeterName(k) 3194
ShephardToddNumber(X, n) 3194
ComplexRootDatum(k) 3195
ComplexRootDatum(m, p, n) 3195

105.6 Operations on Reflection Groups 3196

IsCoxeterIsomorphic(W1, W2) 3196
IsCartanEquivalent(W1, W2) 3196
CartanName(W) 3196
CoxeterDiagram(W) 3196
DynkinDiagram(W) 3196
RootSystem(W) 3197
RootDatum(W) 3197
CoxeterMatrix(W) 3197
CoxeterGraph(W) 3197
CartanMatrix(W) 3197
DynkinDigraph(W) 3197
Rank(W) 3197
NumberOfGenerators(W) 3197
FundamentalGroup(W) 3197
IsogenyGroup(W) 3197
CoisogenyGroup(W) 3198
BasicDegrees(W) 3198
BasicCodegrees(W) 3198
LongestElement(W) 3198
CoxeterElement(W) 3198
CoxeterNumber(W) 3198
LeftDescentSet(W, w) 3199
RightDescentSet(W, w) 3199

105.7 Properties of Reflection Groups 3200

IsReflectionGroup(G) 3200
RootsAndCoroots(G) 3200
IsRealReflectionGroup(G) 3200
IsCrystallographic(W) 3201
IsSimplyLaced(W) 3201
Dual(G) 3201
Overgroup(H) 3201
Overdatum(H) 3201
StandardAction(W) 3201
StandardActionGroup(W) 3201

105.8 Roots, Coroots and Reflections 3202

105.8.1 Accessing Roots and Coroots . . 3202

RootSpace(W) 3202
CorootSpace(W) 3202
SimpleOrders(W) 3202
SimpleRoots(W) 3202
SimpleCoroots(W) 3202
NumberOfPositiveRoots(W) 3202
NumPosRoots(W) 3202
Roots(W) 3202
Coroots(W) 3202
PositiveRoots(W) 3203
PositiveCoroots(W) 3203
Root(W, r) 3203
Coroot(W, r) 3203
RootPosition(W, v) 3203
CorootPosition(W, v) 3203

105.8.2 Reflections 3205

ReflectionMatrices(W) 3205

`CoreflectionMatrices(W)` 3205
`SimpleReflectionMatrices(W)` 3205
`SimpleCoreflectionMatrices(W)` 3205
`ReflectionMatrix(W, r)` 3205
`CoreflectionMatrix(W, r)` 3205
`SimpleReflectionPermutations(W)` 3205
`ReflectionPermutations(W)` 3205
`ReflectionPermutation(W, r)` 3205
`ReflectionWords(W)` 3205
`ReflectionWord(W, r)` 3205
`Length(w)` 3206
`CoxeterLength(w)` 3206
105.8.3 Weights 3206
`WeightLattice(W)` 3206
`CoweightLattice(W)` 3206
`FundamentalWeights(W)` 3206
`FundamentalCoweights(W)` 3206
`IsDominant(R, v)` 3207
`DominantWeight(W, v)` 3207
`WeightOrbit(W, v)` 3207
105.9 Related Structures 3208
`CoxeterGroup(GrpFPCox, W)` 3208
`CoxeterGroup(GrpPermCox, W)` 3208
`LieAlgebra(W, R)` 3208
`GroupOfLieType(W, k)` 3208
105.10 Bibliography 3208

Chapter 105
REFLECTION GROUPS

105.1 Introduction

A reflection is a diagonalisable linear transformation of finite order whose space of fixed points is a hyperplane. A reflection group is a finite dimensional linear group over a field F, which is generated by a finite number of reflections.

There are no restrictions on the field F and there is no requirement for a reflection to be a transformation of order two. However, if F is a real field, every reflection does have order two and there is a much richer theory. In particular, every Coxeter group is a real reflection group (see Chapter 104).

The books [LT09], [Bro10] or [Kan01] are useful references for complex reflection groups. Standard references for the theory of real reflection groups include [Bou68, Chapters 4, 5, 6] and [Hum90].

105.2 Construction of Pseudo-reflections

Let V be a vector space of dimension n over a field F. As defined in Bourbaki [Bou68], a *pseudo-reflection* in MAGMA is a linear transformation of V whose space of fixed points is a subspace of dimension $n-1$, namely a hyperplane. (Some authors require a pseudo-reflection to be invertible and diagonalisable.)

A reflection, as defined above, is a pseudo-reflection and so too is a transvection. The MAGMA package described in this chapter includes code for the construction of transvections but the emphasis is on groups generated by reflections.

If r is a pseudo-reflection, then $\dim(\mathrm{im}(1-r)) = 1$ and a basis element of $\mathrm{im}(1-r)$ is called a *root* of r.

Let a be a root of the pseudo-reflection r and let $H = \ker(1-r)$ be the hyperplane of fixed points of r. For all $v \in V$ there exists $\phi(v) \in F$ such that $v - vr = \phi(v)a$. Then $\phi \in V^*$ and $\ker\phi = H$. This means that every pseudo-reflection has the form

$$vr = v - \phi(v)a$$

and its determinant is $1 - \phi(a)$. The linear functional ϕ is a *coroot* of r.

- If $\phi(a) = 1$, then r is not invertible; it is the *projection* of V onto H along a.
- If $\phi(a) = 0$ (equivalently, $a \in H$), then r is by definition a *transvection*.
- If $\phi(a) \neq 0, 1$, then r is called a *reflection*. For the most part we consider only reflections of finite order, but not necessarily of order two.

In MAGMA both V and its dual space V^* are identified with the space F^n of row vectors of length n and the standard bilinear pairing between V and V^* is $(a, b) \mapsto ab^{\mathrm{tr}}$, where b^{tr} denotes the column vector which is the *transpose* of b.

The row vector b which represents the coroot ϕ is also called a *coroot* of the pseudo-reflection; it is uniquely determined by r and a. The matrix of r is

$$I - b^{\mathrm{tr}}a$$

and, in particular, $ar = (1 - ab^{\mathrm{tr}})a$. Thus r is a reflection of finite order d if and only if $ab^{\mathrm{tr}} \neq 0, 1$ and $1 - ab^{\mathrm{tr}}$ is a d-th root of unity.

`PseudoReflection(a, b)`

The matrix of the pseudo-reflection with root a and coroot b.

`Transvection(a, b)`

The matrix of the transvection with root a and coroot b. The input is checked to ensure that the root and coroot define a transvection.

`Reflection(a, b)`

The matrix of the reflection with root a and coroot b. The input is checked to ensure that the root and coroot define a reflection.

`IsPseudoReflection(r)`

Returns `true` if r is the matrix of a pseudo-reflection, in which case a root and a coroot are returned as well.

`IsTransvection(r)`

Returns `true` if r is the matrix of a transvection, in which case a root and a coroot are returned as well.

`IsReflection(r)`

Returns `true` if r is the matrix of a reflection, in which case a root and a coroot are returned as well.

`IsReflectionGroup(G)`

`Strict` BoolElt *Default :* `true`

The default action is to return `true` if every generator of G is a reflection. If `Strict` is `false`, the function checks if G can be generated by *some* of its reflections, not necessarily those returned by `Generators(G)`.

Example H105E1

Create a pseudo-reflection directly and then check that it is a transvection.

```
> V := VectorSpace(GF(5), 3);
> t := PseudoReflection(V![1,0,0],V![0,1,0]);
> t;
[1 0 0]
[4 1 0]
[0 0 1]
> IsTransvection(t);
true (1 0 0)
(0 1 0)
> IsReflection(t);
false
```

Example H105E2

An example of a group which can be generated by reflections even though not every given generator is a reflection.

```
> F<omega> := CyclotomicField(3);
> r := Matrix(F,2,2,[1,omega^2,0,omega]);
> IsReflection(r);
true (         0 -omega + 1)
(1/3*(2*omega + 1)                  1)
> s := Matrix(F,2,2,[0,-1,1,0]);
> IsReflection(s);
false
> G := MatrixGroup<2,F | r,s >;
> IsReflectionGroup(G);
false
> IsReflectionGroup(G : Strict := false);
true
> #G;
24
```

To find reflection generators for this group we look for a reflection which, together with the reflection r, generates G. (This is a rather special example; not every finite reflection group of rank two can be generated by two reflections.)

```
> exists(t){ t : t in G | IsReflection(t) and G eq sub<G|r,t> };
true
> t;
[        0 omega + 1]
[        1    -omega]
```

Example H105E3

The groups $\mathrm{SL}(n, q)$ are generated by transvections. To illustrate this we find representatives for the conjugacy classes of $\mathrm{GL}(3, 25)$ which are transvections and then check that the normal closure is $\mathrm{SL}(3, 25)$.

```
> G := GL(3,25);
> ccl := Classes(G);
> T := [ c : c in ccl | IsTransvection(c[3]) ];
> #T;
1
> t := T[1][3]; t;
[     1      0      0]
[     0      1      1]
[     0      0      1]
> S := ncl< G | t >;
> S eq SL(3,25);
true
```

105.2.1 Pseudo-reflections Preserving Reflexive Forms

Let J be the matrix of a non-degenerate reflexive bilinear or sesquilinear form β on the vector space V over a field F. Then β is either a symmetric, alternating or hermitian form.

We may assume that F is equipped with an automorphism σ such that $\sigma^2 = 1$. If β is a symmetric or alternating form, σ is the identity; if β is hermitian, the order of $\sigma : \alpha \mapsto \bar{\alpha}$ is two and $J = \bar{J}^{\mathrm{tr}}$. If a is the row vector $(\alpha_1, \alpha_2, \ldots, \alpha_n)$, define $\sigma(a) = (\sigma(\alpha_1), \sigma(\alpha_2), \ldots, \sigma(\alpha_n))$.

If a is a root of a pseudo-reflection r and if r preserves β, then the coroot of r is $\alpha\sigma(a)J^{\mathrm{tr}}$ for some $\alpha \in F$. Thus the matrix of r is $I - \alpha J^{\mathrm{tr}}\sigma(a)^{\mathrm{tr}}a$.

`SymplecticTransvection(a, alpha)`

The symplectic transvection with root a and multiplier α with respect to the form attached to the parent of a. If the form is not alternating a runtime error is generated.

If β is a non-degenerate alternating form preserved by a pseudo-reflection r, then the dimension of V is even and r must be a transvection. If a is a root of r, the coroot is $\alpha a J^{\mathrm{tr}}$ and the matrix of r is $I - \alpha J a^{\mathrm{tr}}a$, for some $\alpha \neq 0$ in F.

`UnitaryTransvection(a, alpha)`

The unitary transvection with root a and multiplier α with respect to the hermitian form attached to the parent of a.

The matrix of the unitary transvection is $I - \alpha J\bar{a}^{\mathrm{tr}}a$, where a is isotropic and the trace of α is 0; that is, $aJ\bar{a}^{\mathrm{tr}} = 0$ and $\alpha + \bar{\alpha} = 0$.

A runtime error is generated if the form is not hermitian, if a is not isotropic, or if the trace of α is not 0.

`UnitaryReflection(a, zeta)`

The unitary reflection with root a and determinant ζ, where ζ is a root of unity. The reflection preserves the hermitian form attached to the ambient space of a and sends a to ζa.

In the case of a unitary reflection r with matrix $I-\alpha J^{\mathrm{tr}}\sigma(a)^{\mathrm{tr}}a$, the root a must be non-isotropic and $ar = \zeta a$, where ζ is a root of unity. Therefore, $\alpha = (1-\zeta)/aJ\bar{a}^{\mathrm{tr}}$.

The vector $a^\vee = \bar{\alpha}a$ is the *coroot* of a and the definition of r becomes

$$vr = v - \beta(v, a^\vee)a.$$

`OrthogonalReflection(a)`

The reflection determined by a non-singular vector a of a quadratic space.

A *quadratic space* is a vector space V equipped with a quadratic form Q (see Chapter 93 for more details). The *polar form* of Q is the symmetric bilinear form $\beta(u, v) = Q(u+v) - Q(u) - Q(v)$. Thus $\beta(v, v) = 2Q(v)$ and therefore, if the characteristic of F is not two, Q is uniquely determined by β.

If a is non-singular (that is, $Q(a) \neq 0$), the formula

$$vr = v - Q(a)^{-1}\beta(v, a)a$$

defines a pseudo-reflection. If the characteristic of F is 2, this is a transvection; in all other cases it is a reflection. However, in characteristic 2 there is a certain ambivalence in the literature and the pseudo-reflections just defined are often called reflections.

The *coroot* of a is $a^\vee = Q(a)^{-1}a$. If the characteristic of F is not two, then $a^\vee = 2a/\beta(a, a)$ and this coincides with the usual notion of coroot, as found in [Hum90], for example. In particular, if $\beta(u, v)$ is the standard inner product $(u, v) = uv^{\mathrm{tr}}$, then the inner product and the pairing between V and its dual are essentially the same and the concepts of coroot and coroot coincide.

Example H105E4

We create an hermitian space by attaching an hermitian form J to a vector space V over a field with complex conjugation. The vector $a = (1, 0, 0, 0)$ is isotropic with respect to this form and therefore we can use it to create a unitary transvection.

```
> K<i> := CyclotomicField( 4 );
> sigma := hom< K -> K | x :-> ComplexConjugate(x) >;
> J := Matrix(4,4,[K|0,0,0,1, 0,0,1,0, 0,1,0,0, 1,0,0,0]);
> V := UnitarySpace(J,sigma);
> a := V![1,0,0,0];
> t := UnitaryTransvection(a,i);
> t;
[ 1  0  0  0]
[ 0  1  0  0]
[ 0  0  1  0]
```

```
[-i  0  0  1]
```

Continuing the previous example we note that $b = (1, 1, 1, 1)$ is non-isotropic and we create a unitary reflection of order 4 with b as root.

```
> b := V![1,1,1,1];
> InnerProduct(b,b);
4
> r := UnitaryReflection(b,i);
> r, Eigenvalues(r);
[1/4*(i + 3) 1/4*(i - 1) 1/4*(i - 1) 1/4*(i - 1)]
[1/4*(i - 1) 1/4*(i + 3) 1/4*(i - 1) 1/4*(i - 1)]
[1/4*(i - 1) 1/4*(i - 1) 1/4*(i + 3) 1/4*(i - 1)]
[1/4*(i - 1) 1/4*(i - 1) 1/4*(i - 1) 1/4*(i + 3)]
{
    <i, 1>,
    <1, 3>
}
```

105.3 Construction of Reflection Groups

In MAGMA a *pseudo-reflection group* is a group generated by a finite set of invertible pseudo-reflections. A convenient way to provide the generators for a pseudo-reflection group W is via a finite collection of roots and coroots. In this context the roots and coroots of the generators are called the *basic roots* and *basic coroots* of W.

In the most general case, even when the pseudo-reflection group W is generated by reflections, there are no known distinguished generating reflections whose roots have properties analogous to simple roots in Weyl groups or Coxeter groups. Therefore, one should be careful to distinguish between the basic roots as defined here and the simple (or fundamental) roots of real reflection groups

See Section 105.4 for the construction of real reflection groups and Section 105.5 for the construction of finite complex reflection groups.

`PseudoReflectionGroup(A, B)`

The pseudo-reflection group with the basic roots and corresponding coroots given by the rows of the matrices A and B.

Example H105E5

A direct construction of the Shephard and Todd group $G(14, 1, 2)$ with user supplied roots and coroots.

```
> F<z> := CyclotomicField(7);
> A := Matrix(F,2,3,[[z,0,1],[0,1,0]]);
> B := Matrix(F,2,3,[[1,1,1],[1,2,1]]);
> G<x,y> := PseudoReflectionGroup(A,B);
> IsReflectionGroup(G);
```

```
true
> Order(x),Order(y),Order(x*y);
14 2 28
> #G;
392
```

105.4 Construction of Real Reflection Groups

The only root of unity in the real field is -1, hence every pseudoreflection over the real field is a reflection. We call a reflection group *real* if it is defined over the reals and its simple roots and simple coroots are linearly independent. We allow real reflection groups to be defined as matrix groups over the integer ring (Chapter 18), the rational field (Chapter 20), number fields (Chapter 35), and cyclotomic fields (Chapter 37); the real field (Chapter 25) is *not* allowed since it is not infinite precision.

The real reflection groups are just the reflection representations of the Coxeter groups (Chapter 104). This allows us to compute many more properties for these groups than for general reflection groups. Note that the classification of finite real reflection groups is given in Section 101.6.

`ReflectionGroup(M)`
`ReflectionGroup(G)`
`ReflectionGroup(C)`
`ReflectionGroup(D)`

The reflection group with Coxeter matrix M, Coxeter graph G, Cartan matrix C, or Dynkin digraph D (see Chapter 101).

`ReflectionGroup(N)`

The finite or affine reflection group with Cartan name given by the string N (see Section 101.6).

`IrreducibleReflectionGroup(X, n)`

The finite or affine irreducible reflection group with Cartan name X_n (see Section 101.6).

Example H105E6

```
> C := CartanMatrix("B3" : Symmetric);
> G := ReflectionGroup(C);
> G;
MatrixGroup(3, Number Field with defining polynomial x^2 - 2 over the Rational
Field) of order 48 = 2^4 * 3
Generators:
    [-1  0  0]
    [ 1  1  0]
    [ 0  0  1]

    [  1   1   0]
    [  0  -1   0]
    [  0 $.1   1]

    [  1   0   0]
    [  0   1 $.1]
    [  0   0  -1]
```

`ReflectionGroup(R)`

The finite reflection group with root system or root datum R (see Chapters 102 and 103).

Example H105E7

```
> R := RootDatum("B3");
> ReflectionGroup(R);
MatrixGroup(3, Integer Ring) of order 48 = 2^4 * 3
Generators:
    [-1  0  0]
    [ 1  1  0]
    [ 0  0  1]

    [ 1  1  0]
    [ 0 -1  0]
    [ 0  1  1]

    [ 1  0  0]
    [ 0  1  2]
    [ 0  0 -1]
```

`ReflectionGroup(W)`

A	MTRX	*Default :*
B	MTRX	*Default :*
C	MTRX	*Default :*

A reflection group W' of the Coxeter group W, together with the isomorphism $W \to W'$ (see Chapter 104). Since a Coxeter group W does not come with an in-built reflection representation, the optional parameters A, B, and C can be used to specify the representation. They are respectively the matrix whose rows are the simple roots, the matrix whose rows are the simple coroots, and the Cartan matrix. These must have the following properties:

1. A and B must have same number of rows and the same number of columns; they must be defined over the same field, which must be the rational field, a number field, or a cyclotomic field; the entries must be real;
2. the number of columns must be at least the number of rows; and
3. $C = AB^{\mathrm{tr}}$ must be a Cartan matrix for W.

It is not necessary to specify all three matrices: any two of them will determine the third. If C is not determined, it is taken to be the standard matrix described in Section 101.4.

`ReflectionGroup(W)`

The reflection group W' isomorphic to the permutation Coxeter group W, together with the isomorphism $W \to W'$ (see Chapter 104). There are no optional parameters A, B, and C in this case because every permutation Coxeter group has a root system, and this determines the reflection representation.

Example H105E8

```
> W<a,b,c> := CoxeterGroup(GrpFPCox, "B3");
> G, h := CoxeterGroup(GrpMat, W);
> a*b; h(a*b);
a * b
[-1 -1  0]
[ 1  0  0]
[ 0  1  1]
```

105.5 Construction of Finite Complex Reflection Groups

In this section, we describe the classification and construction of finite complex reflection groups.

A finite complex reflection group has a finite root system but there is no known analogue of a set of simple roots as in the theory of finite Coxeter groups. To illustrate the difficulty, one of the examples in this section constructs a complex reflection group of rank 4 which cannot be generated by fewer than 5 generators.

Nevertheless, it is possible to generalise the concept of *root datum* to the complex case and construct all complex reflection groups via their root data.

Let D be the ring of integers of a number field F which admits a well-defined operation of complex conjugation (which in the case of a real number field will be the identity automorphism). Let $\mu(D)$ be the group of roots of unity in D and let $V = F \times_D L$.

A *complex root datum* is a 4-tuple (L, L^*, Φ, ρ), where

- L and L^* are free D-modules of rank n which are in duality via a pairing $L \times L^* \to D : (a, \phi) \mapsto \langle\, a, \phi \,\rangle$;
- Φ is a finite subset of L and $\rho : \Phi \to L^*$.

For all $a \in \Phi$ we have:

1. for all $\lambda \in F$, we have $\lambda a \in \Phi$ if and only if $\lambda \in \mu(D)$;
2. for all $\lambda \in D$, we have $\rho(\lambda a) = \overline{\lambda}\rho(a)$;
3. $f(a) = 1 - \langle\, a, \rho(a) \,\rangle \in \mu(D) \setminus \{1\}$;
4. the reflection r_a of V defined by $vr_a = v - \langle\, v, \rho(a) \,\rangle a$ and the reflection r_a^* of V^* defined by $\phi r_a^* = \phi - \langle\, a, \phi \,\rangle \rho(a)$ satisfy:
   - $\Phi r_a \subseteq \Phi$ and $\Phi^* r_a^* \subseteq \Phi^*$, where $\Phi^* = \rho(\Phi)$.
   - $f(ar_b) = f(a)$ for all $a, b \in \Phi$.

Put $a^* = \rho(a)$ and $V^* = F \otimes_D L^*$. Then $\rho : \Phi \to \Phi^*$ is a bijection and the map

$$p := V \to V^* : v \mapsto \sum_{a \in \Phi} \overline{\langle\, v, a^* \,\rangle} a^*$$

is semilinear. Furthermore, $\beta(u, v) = \langle\, u, p(v) \,\rangle$ defines a non-degenerate hermitian form on the span of Φ.

The group W generated by the reflections $\{r_a \mid a \in \Phi\}$ is the *Weyl group* of the root datum. For any set $\{r_1, r_2, \ldots, r_k\}$ of reflections that generate W, every reflection in W is conjugate to a power of some r_i. The set Φ is a *root system* for W and Φ^* is the set of *coroots*.

If $a_1, a_2, \ldots, a_k \in \Phi$ are roots of the reflections $r_1, r_2, \ldots, r_n$ which generate W, then $C = (\langle\, a_i, a_j^* \,\rangle)$ is a *complex Cartan matrix* and the a_i and a_j^* are *basic* roots and coroots of W.

Even though there is no satisfactory notion of 'simple roots', a complex reflection group can nevertheless be described by means of a complex Cartan matrix. In MAGMA if the roots are the rows of a matrix A and if the coroots are the rows of a matrix B, then $C = AB^{\mathrm{tr}}$. The matrices A and B are called *basic* root and coroot matrices.

The complex Cartan matrix can be described by a diagram similar to the Dynkin diagram of a Coxeter group. This notation was suggested by Coxeter and used by Cohen in [Coh76]. (There is a different type of diagram used by Broué, Malle and others.)

Cohen's naming scheme for the diagrams extends the standard notation A_n, B_n, ..., H_3, H_4 used for Coxeter groups. MAGMA uses a slight variation of Cohen's scheme; that is, in MAGMA, Cohen's group EN_4 is referred to as O_4.

The original numbering system for the primitive complex reflection groups is due to Shephard and Todd [ST54].

`ShephardTodd(n)`

`NumFld` BOOLELT *Default :* `false`

This function returns the primitive reflection group G_n using the Shephard and Todd numbering.

By default the matrices are written over the ring of integers of the smallest cyclotomic field which contains the character values of the reflections. If the parameter `NumFld` is set to `true`, the number field generated by the character values of the reflections is used.

The groups available via this function include all finite primitive complex reflection groups other than the symmetric groups Sym(n) for $n \geq 5$. The groups are listed below.

Nineteen 2-dimensional primitive complex reflection groups:

Tetrahedral family: G_4, ..., G_7
Octahedral family: G_8, ..., G_{15}
Icosahedral family: G_{16}, ..., G_{22}

Five 3-dimensional complex reflection groups:

G_{23}: $W(H_3) = \mathbf{Z}_2 \times PSL(2,5)$, order 120.
G_{24}: $W(J_3(4)) = \mathbf{Z}_2 \times PSL(2,7)$, order 336.
G_{25}: $W(L_3) = 3^{1+2} \cdot SL(2,3)$, order 648; Hessian group.
G_{26}: $W(M_3) = \mathbf{Z}_2 \times 3^{1+2} \cdot SL(2,3)$, order 1296; Hessian group.
G_{27}: $W(J_3(5)) = \mathbf{Z}_2 \times (\mathbf{Z}_3 \cdot \mathrm{Alt}(6))$, order 2160, where $\mathbf{Z}_3 \cdot \mathrm{Alt}(6)$ denotes the non-split extension of $\mathbf{Z}_3$ by Alt(6).

Five 4-dimensional complex reflection groups in addition to Sym(5):

G_{28}: $W(F_4) = (SL(2,3) \circ SL(2,3)) \cdot (\mathbf{Z}_2 \times \mathbf{Z}_2)$, order 1152.
G_{29}: $W(N_4) = (\mathbf{Z}_4 \circ 2^{1+4}) \cdot \mathrm{Sym}(5)$, order 7680 (splits).
G_{30}: $W(H_4) = (SL(2,5) \circ SL(2,5)) \cdot \mathbf{Z}_2$, order 14 400.
G_{31}: $W(O_4) = (\mathbf{Z}_4 \circ 2^{1+4}) \cdot Sp(4,2)$, order 46 080 (non-split) 5 generators.
G_{32}: $W(L_4) = \mathbf{Z}_3 \times Sp(4,3)$, order $155\,520 = 2^7 \times 3^5 \times 5$.

One 5-dimensional complex reflection group in addition to Sym(6):

G_{33}: $W(K_5) = \mathbf{Z}_2 \times \Omega(5,3) = \mathbf{Z}_2 \times PSp(4,3) = \mathbf{Z}_2 \times PSU(4,2)$, order $51\,840 = 2^7 \times 3^4 \times 5$.

Two 6-dimensional complex reflection groups in addition to Sym(7):

G_{34}: $W(K_6) = \mathbf{Z}_3 \cdot \widehat{\Omega}^-(6,3)$, order $39\,191\,040 = 2^9 \times 3^7 \times 5 \times 7$ (non-split), where $\widehat{\Omega}^-(6,3)$ is a semidirect product of $\Omega^-(6,3)$ by $\mathbf{Z}_2$.

G_{35}: $W(E_6) = SO(5,3) = O^-(6,2) = PSp(4,3) \cdot \mathbf{Z}_2 = PSU(4,2) \cdot \mathbf{Z}_2$, order $51\,840 = 2^7 \times 3^4 \times 5$.

One 7-dimensional complex reflection group in addition to Sym(8):

G_{36}: $W(E_7) = \mathbf{Z}_2 \times Sp(6,2)$, order $2\,903\,040 = 2^{10} \times 3^4 \times 5 \times 7$.

One 8-dimensional complex reflection group in addition to Sym(9):

G_{37}: $W(E_8) = \mathbf{Z}_2 \cdot O^+(8,2)$, order $696\,729\,600 = 2^{14} \times 3^5 \times 5^2 \times 7$ (non-split).

Example H105E9

We verify that the complex reflection group G_{24} is isomorphic to $\mathbf{Z}_2 \times \Omega(3,7)$.

```
> W := ShephardTodd(24);
> G := sub<GL(3,7) | Omega(3,7), -GL(3,7)!1>;
> IsIsomorphic(W,G):Minimal;
true Homomorphism of MatrixGroup(3, Cyclotomic Field of order 7 and degree 6)
of order 2^4 * 3 * 7 into MatrixGroup(3, GF(7)) of order 2^4 * 3 * 7
```

`ComplexReflectionGroup(C)`

`Reduced` BoolElt *Default :* `true`

This function returns the complex reflection group defined by the (complex) Cartan matrix C. When the optional parameter `Reduced` is `true` (the default), the roots and coroots are computed modulo the null space of C.

`ComplexReflectionGroup(X, n)`

`NumFld` BoolElt *Default :* `false`

This function returns the primitive reflection group of type X and rank n, using the Cohen/Coxeter naming scheme.

By default the matrices are written over the ring of integers of the smallest cyclotomic field which contains the character values of the reflections. If the parameter `NumFld` is set to `true`, the number field generated by the character values of the reflections is used.

Example H105E10

In this example we find (up to conjugacy) all subgroups of $G = W(O_4) = G_{31}$ that are generated by reflections. This shows that G cannot be generated by fewer than 5 reflections.
We begin by checking that G has only one class of reflections.

```
> G := ComplexReflectionGroup("O",4);
> print #[c[3] : c in Classes(G) | IsReflection(c[3])];
1
> R := Class(G,G.1); #R;
60
> #G;
46080
```

We proceed by building the list of reflection subgroups in 'layers', where the n-th layer consists of representatives of the subgroups generated by n reflections.

```
> L := [sub<G|G.1>];
> layers := [L];
> n := 0;
> while true do
>   n +:= 1;
>   nextlayer := [];
```

We extend each group in layer n by adjoining one additional reflection. The resulting subgroup will be generated by n or $n+1$ reflections. If we haven't seen it before we add it to the list.

```
>   for H in layers[n] do
>     for A in {sub<G|H,s> : s in R | s notin H} do
>       if forall{B : B in L | not IsConjugate(G,A,B)} then
>         Append(~nextlayer,A);
>         Append(~L,A);
>       end if;
>     end for;
>   end for;
>   if IsEmpty(nextlayer) then break; end if;
>   Append(~layers,nextlayer);
```

After the construction of each layer we print the orders of the subgroups.

```
>   print n+1,"generators";
>   print [#A : A in nextlayer];
> end while;
2 generators
[ 4, 4, 8, 6 ]
3 generators
[ 8, 8, 16, 24, 12, 16, 24, 48, 16, 96 ]
4 generators
[ 192, 16, 32, 96, 16, 192, 32, 48, 1536, 384, 64, 64, 32,
 7680, 1152, 36, 384, 120, 120, 192, 192 ]
5 generators
```

```
[ 64, 384, 3072, 128, 46080 ]
6 generators
[ 256 ]
```

Looking at the orders we see that the first time the group G_{31} appears is in layer 5. That is, it cannot be generated by 4 or fewer reflections. It is interesting to note that there is one subgroup which requires 6 generators; namely the imprimitive group $G(4,2,2) \times G(4,2,2)$.

`ShephardTodd(m, p, n)`

`ImprimitiveReflectionGroup(m, p, n)`

`NumFld` BoolElt *Default :* `false`

Let B be the direct product of n copies of the cyclic group C_m of order m and represent the elements of B by diagonal matrices $\mathrm{diag}(\theta_1, \theta_2, \ldots, \theta_n)$. The elements of the symmetric group $\mathrm{Sym}(n)$ can be represented by $n \times n$ permutation matrices and in this guise it acts on the group B; the resulting semidirect product is also known as the *wreath product* $C_m \wr \mathrm{Sym}(n)$.

For each divisor p of m define

$$A(m,p,n) := \{\, \mathrm{diag}(\theta_1, \theta_2, \ldots, \theta_n) \in B \mid (\theta_1\theta_2 \cdots \theta_n)^{m/p} = 1 \,\}.$$

It is immediately clear that $A(m,p,n)$ is a subgroup of index p in B that is invariant under the action of $\mathrm{Sym}(n)$. The semidirect product of $A(m,p,n)$ by the symmetric group $\mathrm{Sym}(n)$ is the group $G(m,p,n)$. These groups are imprimitive when $m \geq 2$. The group $G(1,1,n)$ is the symmetric group $\mathrm{Sym}(n)$ acting as permutation matrices.

Shephard and Todd proved that every irreducible imprimitive complex reflection subgroup of $GL(n, \mathbf{C})$ is conjugate to $G(m,p,n)$ for some m and p.

This function returns the Shephard and Todd group $G(m,p,n) \subset \mathrm{GL}(n,F)$, where p divides m. In general, $G(m,p,n)$ is irreducible but if $m = p = 1$, the function returns $\mathrm{Sym}(n)$ in its natural permutation representation, which is not irreducible.

By default the matrices are written over the ring of integers of the smallest cyclotomic field which contains the character values of the reflections. If the parameter `NumFld` is set to `true`, the number field generated by the character values of the reflections is used.

Example H105E11

```
> ShephardTodd(6, 3, 3);
MatrixGroup(3, Cyclotomic Field of order 6 and degree 2)
Generators:
     [0 1 0]
     [1 0 0]
     [0 0 1]
```

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

[1      0      0]
[0      0      z]
[0 -z + 1      0]

[1  0   0]
[0  1   0]
[0  0  -1]
Mapping from: MatrixGroup(3, Cyclotomic Field of order 6 and degree 2) to GL(3,
CyclotomicField(6))
```

`ComplexRootMatrices(k)`

`ComplexRootMatrices(m, p, n)`

`NumFld` BoolElt *Default :* `false`

If G is the complex reflection group `ShephardTodd(k)` or `ShephardTodd(m,p,n)`, respectively, these functions return five values: the basic root and coroot matrices for G, an invariant hermitian form, a generator for the group of roots of unity of the ring of definition, and the order of the generator.

By default the root matrices are written over the ring of integers of the smallest cyclotomic field which contains the character values of the reflections. If the parameter `NumFld` is set to `true`, the number field generated by the character values of the reflections is used.

Example H105E12

```
> A,B,J,gen,ordgen := ComplexRootMatrices(13);
> A,B;
[            1               0]
[ -z^2 - z - 1               1]
[z^2 + 2*z + 1    -z^2 - z - 1]

[             2     -z^3 + z + 2]
[           z^2    z^3 + z^2 + 1]
[-z^3 + z^2 - 1          z^2 - 1]
> gen,ordgen;
-z^3
8
> G := PseudoReflectionGroup(A,B);
> #G;
96
```

`ComplexCartanMatrix(k)`

`ComplexCartanMatrix(m, p, n)`

`NumFld` BoolElt *Default :* `false`

If A and B are the basic root and coroot matrices returned by `ComplexRootMatrices` above, then this function returns AB^{tr}, where B^{tr} is the transpose of B. The meaning of the optional parameter `NumFld` has been described above.

`BasicRootMatrices(C)`

`Reduced` BoolElt *Default :* `true`

This function returns a matrix A of roots and a matrix B of coroots such that $C = AB^{\mathrm{tr}}$. The default, when the optional parameter `Reduced` is `true`, is to compute the roots and coroots modulo the null space of C.

`CohenCoxeterName(k)`

Cohen's string name and rank of the Shephard and Todd group G_k. This is an extension of the naming scheme for Coxeter groups. For example, the Shephard and Todd group G_{37} is the Coxeter group of type E_8 whereas the Shephard and Todd group G_{32} has Cohen name L_4.

`ShephardToddNumber(X, n)`

Given a string X and an integer n, this function returns the Shephard and Todd number of the complex reflection group $W(X_n)$ of type X and rank n. The rank is the dimension of the space on which the group acts; it is not always the number of generators.

The Shepard and Todd numbers range from 1 to 37. All symmetric groups (type A) have Shephard and Todd number 1, all imprimitive groups $G(m, p, n)$ have Shephard and Todd number 2, and all cyclic groups have Shephard and Todd number 3. The primitive complex reflection groups of rank 2 have Shepard and Todd numbers in the range 4 to 22. Except for the group G_4 which has type L_2, the rank 2 groups do not have Cohen–Coxeter names.

The Shephard and Todd numbers in the range 23 to 37 refer to the Cohen–Coxeter groups $W(E_6)$, $W(E_7)$, $W(E_8)$, $W(F_4)$, $W(H_3)$, $W(H_4)$, $W(J_3(4))$, $W(J_3(5))$, $W(K_5)$, $W(K_6)$, $W(L_3)$, $W(L_4)$, $W(M_3)$, $W(N_4)$, and $EW(N_4)$. Note that in Magma the types of the rank 3 groups $W(J_3(4))$ and $W(J_3(5))$ are $J4$ and $J5$; and the type of the rank 4 group $EW(N_4)$ is O.

As a matrix group the Coxeter group of type A is returned by the function `CoxeterGroup(GrpMat,"A",n)`, where n is the rank. The groups of types B, C and D are Coxeter groups and imprimitive complex reflection groups. Thus, as matrix groups, they can be obtained via the function `ShephardTodd(2,p,n)`, where $p = 1$ for type B or C and $p = 2$ for type D.

Example H105E13

The type of the group G_{31} is O and its rank is 4. This is the notation used in [LT09].

```
> ShephardToddNumber("J5",3);
27
> CohenCoxeterName(31);
O 4
```

Example H105E14

To construct a complex reflection group with a given name, first convert the name to its Shephard and Todd number.

```
> G := ShephardTodd(ShephardToddNumber("L",4));
> G;
MatrixGroup(4, Cyclotomic Field of order 3 and degree 2)
Generators:
    [     omega           0           0           0]
    [-omega - 1           1           0           0]
    [         0           0           1           0]
    [         0           0           0           1]

    [        1 omega + 1           0           0]
    [        0     omega           0           0]
    [        0 omega + 1           1           0]
    [        0           0           0           1]

    [         1           0           0           0]
    [         0           1 -omega - 1           0]
    [         0           0      omega           0]
    [         0           0 -omega - 1           1]

    [        1           0           0           0]
    [        0           1           0           0]
    [        0           0           1 omega + 1]
    [        0           0           0     omega]
```

`ComplexRootDatum(k)`

`ComplexRootDatum(m, p, n)`

`NumFld` BoolElt *Default :* `false`

A root datum for the Shephard and Todd group G_k or, in the second form of the function, the imprimitive group $G(m, p, n)$. This is returned as a 5-tuple Φ, Φ^*, ρ, W, J, where Φ is the sequence of roots, Φ^* the sequence of coroots, $\rho : A \to B$ is a bijective map, W is the complex reflection group of the root datum, and J is an hermitian form preserved by W.

105.6 Operations on Reflection Groups

See Chapter 64 for general functions for matrix groups. Note that most of the functions in this section only work for real reflection groups.

`IsCoxeterIsomorphic(W1, W2)`

Returns `true` if and only if the real reflection groups W_1 and W_2 are isomorphic as Coxeter systems.

`IsCartanEquivalent(W1, W2)`

Returns `true` if and only if the crystallographic real reflection groups W_1 and W_2 have Cartan equivalent Cartan matrices.

Example H105E15

```
> W1 := ReflectionGroup("B3");
> W2 := ReflectionGroup("C3");
> IsCoxeterIsomorphic(W1, W2);
true [ 1, 2, 3 ]
> IsCartanEquivalent(W1, W2);
false
```

`CartanName(W)`

The Cartan name of the finite or affine real reflection group W (Section 101.6).

`CoxeterDiagram(W)`

A display of the Coxeter diagram of the real reflection group W (Section 101.6). If W is not affine or finite, an error is flagged.

`DynkinDiagram(W)`

A display of the Coxeter diagram of the real reflection group W (Section 101.6). If W is not affine or finite, or if W is not crystallographic, an error is flagged.

Example H105E16

```
> G := CompleteGraph(3);
> W := ReflectionGroup(G);
> CartanName(W);
A~2
> CoxeterDiagram(W);
A~2    1 - 2
       |   |
       - 3 -
```

`RootSystem(W)`

The root system of the finite real reflection group W (Chapter 102). If W is infinite, an error is flagged.

`RootDatum(W)`

The root datum of the finite real reflection group W (Chapter 103). The roots and coroots of W must have integral components, and W must be finite.

`CoxeterMatrix(W)`

The Coxeter matrix of the real reflection group W (Section 101.2).

`CoxeterGraph(W)`

The Coxeter graph of the real reflection group W (Section 101.3).

`CartanMatrix(W)`

The Cartan matrix of the real reflection group W (Section 101.4).

`DynkinDigraph(W)`

The Dynkin digraph of the real reflection group W (Section 101.5).

`Rank(W)`
`NumberOfGenerators(W)`

The rank of the reflection group W.

Example H105E17

```
> R := StandardRootSystem("A", 4);
> W := ReflectionGroup(R);
> Rank(W);
4
> Dimension(W);
5
```

`FundamentalGroup(W)`

The fundamental group of the real reflection group W (Subsection 103.1.6). The roots and coroots of W must have integral components.

`IsogenyGroup(W)`

The isogeny group of the real reflection group W, together with the injection into the fundamental group (Subsection 103.1.6). The roots and coroots of W must have integral components.

`CoisogenyGroup(W)`

The fundamental group of the real reflection group W together with the projection onto the fundamental group (Subsection 103.1.6). The roots and coroots of W must have integral components.

`BasicDegrees(W)`

The degrees of the basic invariant polynomials of the reflection group W. These are computed using the table in [Car72, page 155] if the group is real, and using the algorithm of [LT09] in other cases. If W is infinite, an error is flagged.

`BasicCodegrees(W)`

The basic codegrees of the reflection group W. These are computed using the algorithm of [LT09]. If W is infinite, an error is flagged.

Example H105E18

The product of the basic degrees is the order of the Coxeter group; the sum of the basic degrees is the sum of the rank and the number of positive roots.

```
> W := ReflectionGroup("E6");
> degs := BasicDegrees(W);
> degs;
[ 2, 5, 6, 8, 9, 12 ]
> &*degs eq #W;
true
> &+degs eq NumPosRoots(W) + Rank(W);
true
```

`LongestElement(W)`

The unique longest element in the finite real reflection group W.

`CoxeterElement(W)`

The Coxeter element in the reflection group W, ie. the product of the generators.

`CoxeterNumber(W)`

The order of the Coxeter element in the real reflection group W.

Example H105E19

Operations on groups.

```
> W := ReflectionGroup("A4");
> LongestElement(W);
[ 0  0  0 -1]
[ 0  0 -1  0]
[ 0 -1  0  0]
[-1  0  0  0]
> CoxeterElement(W);
[-1 -1 -1 -1]
[ 1  0  0  0]
[ 0  1  0  0]
[ 0  0  1  0]
```

`LeftDescentSet(W, w)`

The set of indices r of simple roots of the finite real reflection group W such that the length of the product $s_r w$ is less than that of the element w.

`RightDescentSet(W, w)`

The set of indices r of simple roots of the finite real reflection group W such that the length of the product $w s_r$ is less than that of the element w.

Example H105E20

```
> W := ReflectionGroup("A5");
> x := W.1*W.2*W.4*W.5;
> LeftDescentSet(W, x);
{ 1, 4 }
> RightDescentSet(W, x);
{ 2, 5 }
```

105.7 Properties of Reflection Groups

See Chapter 64 for general functions for matrix groups.

`IsReflectionGroup(G)`

`Strict` BoolElt *Default :* `true`

The default action is to return `true` if every generator of G is a reflection. If `Strict` is `false`, the function checks if G can be generated by *some* of its reflections, not necessarily those returned by `Generators(G)`.

`RootsAndCoroots(G)`

Returns the orders of the reflections, the roots and the coroots of the reflection group G.

`IsRealReflectionGroup(G)`

Returns `true` if and only if the matrix group G is a real reflection group. If `true`, the simple orders, roots, and coroots are also returned.

Example H105E21

```
> W := ComplexReflectionGroup("A", 4);
> IsReflectionGroup(W);
true
> IsRealReflectionGroup(W);
true

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

[ 2 -1  0  0]
[-1  2 -1  0]
[ 0 -1  2 -1]
[ 0  0 -1  2]
> W := ComplexReflectionGroup("M", 3);
> IsReflectionGroup(W);
true
>  IsRealReflectionGroup(W);
                          ^
Runtime error in 'IsRealReflectionGroup': The group must be defined over the
reals
```

`IsCrystallographic(W)`

Returns `true` if and only if the real reflection group W is crystallographic; i.e., its Cartan matrix has integral entries.

`IsSimplyLaced(W)`

Returns `true` if and only if the real reflection group W is simply laced; i.e., its Coxeter graph has no labels.

Example H105E22

```
> W := ReflectionGroup("A~2 D4");
> IsFinite(W);
false
> IsCrystallographic(W);
true
> IsSimplyLaced(W);
true
```

`Dual(G)`

The dual of the reflection group G, ie, the reflection group gotten by swapping roots with coroots.

`Overgroup(H)`

The overgroup of H, ie. the reflection group whose roots are permuted by the elements of the reflection subgroup H.

`Overdatum(H)`

The root datum whose roots are permuted by the elements of the reflection subgroup H.

Every Coxeter group W has a standard action. For example, the standard action group of a Coxeter group of type A_n is the symmetric group of degree $n+1$ acting on $\{1, \ldots, n\}$.

`StandardAction(W)`

The standard action of the reflection group W.

`StandardActionGroup(W)`

The group G of the standard action of the reflection group W, together with an isomorphism $W \to G$.

105.8 Roots, Coroots and Reflections

Many of these functions have an optional argument `Basis` which may take one of the following values

1. `"Standard"`: the standard basis for the (co)root space. This is the default.
2. `"Root"`: the basis of simple (co)roots.
3. `"Weight"`: the basis of fundamental (co)weights (see Subsection 105.8.3 below).

105.8.1 Accessing Roots and Coroots

`RootSpace(W)`

`CorootSpace(W)`

The base space of the reflection group W. If W is not a reflection group, an error occurs.

Example H105E23

```
> W := ComplexReflectionGroup("M", 3);
> RootSpace(W);
Full Vector space of degree 3 over Cyclotomic Field of order 24 and degree 8
```

`SimpleOrders(W)`

The sequence of simple orders of the reflection group W. If W is not a reflection group, an error is flagged.

`SimpleRoots(W)`

`SimpleCoroots(W)`

The simple (co)roots of the reflection group W as the rows of a matrix, i.e. A (resp. B).

`NumberOfPositiveRoots(W)`

`NumPosRoots(W)`

The number of positive roots of the real reflection group W. This is also the number of positive coroots. The total number of (co)roots is twice the number of positive (co)roots. This number is finite if and only if W is finite.

`Roots(W)`

`Coroots(W)`

`Basis` MonStgElt *Default : "Standard"*

The indexed set of (co)roots of the real reflection group W, i.e. $\{@\,\alpha_1, \ldots \alpha_{2N}\,@\}$ (resp. $\{@\,\alpha_1^\star, \ldots \alpha_{2N}^\star\,@\}$). If W is infinite, an error is flagged.

`PositiveRoots(W)`

`PositiveCoroots(W)`

`Basis` MonStgElt *Default :* "*Standard*"

The indexed set of positive (co)roots of the real reflection group W, that is, $\{@\,\alpha_1, \ldots \alpha_N\,@\}$ (resp. $\{@\,\alpha_1^\star, \ldots \alpha_N^\star\,@\}$). If W is infinite, an error is flagged.

`Root(W, r)`

`Coroot(W, r)`

`Basis` MonStgElt *Default :* "*Standard*"

The rth (co)root α_r (resp. $\alpha_r^\star$) of the real reflection group W. If W is infinite, an error is flagged.

`RootPosition(W, v)`

`CorootPosition(W, v)`

`Basis` MonStgElt *Default :* "*Standard*"

If v is a (co)root in the finite real reflection group W, return its index; otherwise return 0. These functions will try to coerce v into the appropriate lattice; v should be written with respect to the basis specified by the parameter `Basis`. If W is infinite, an error is flagged.

Example H105E24

```
> W := ReflectionGroup("A3");
> Roots(W);
{@
    (1 0 0),
    (0 1 0),
    (0 0 1),
    (1 1 0),
    (0 1 1),
    (1 1 1),
    (-1  0  0),
    (0 -1  0),
    (0  0 -1),
    (-1 -1  0),
    (0 -1 -1),
    (-1 -1 -1)
@}
> PositiveCoroots(W);
{@
    (2 -1  0),
    (-1  2 -1),
    (0 -1  2),
    (1  1 -1),
```

```
    (-1  1  1),
    (1 0 1)
@}
> #Roots(W) eq 2*NumPosRoots(W);
true
> Root(W, 4);
(1 1 0)
> Root(W, 4 : Basis := "Root");
(1 1 0)
> RootPosition(W, [1,1,0]);
4
> A := Matrix(3,3,[1,0,0, -1,-1,-3, 1,2,4]);
> B := Matrix(3,3,[2,-1,0, -1,2,-1, 0,1,0]);
> W := ReflectionGroup(A,B);
> Roots(W);
{@
    (1 0 0),
    (-1 -1 -3),
    (1 2 4),
    (0 -1 -3),
    (0 1 1),
    (1 1 1),
    (-1  0  0),
    (1 1 3),
    (-1 -2 -4),
    (0 1 3),
    (0 -1 -1),
    (-1 -1 -1)
@}
> PositiveCoroots(W);
{@
    (2 -1  0),
    (-1  2 -1),
    (0 1 0),
    (1  1 -1),
    (-1  3 -1),
    (1  2 -1)
@}
> #Roots(W) eq 2*NumPosRoots(W);
true
> Root(W, 4);
(0 -1 -3)
> Root(W, 4 : Basis := "Root");
(1 1 0)
> RootPosition(W, [0,-1,-3]);
4
```

105.8.2 Reflections

The root α acts on the root space via the reflection s_α; the coroot $\alpha^\star$ acts on the coroot space via the coreflection $s_\alpha^\star$.

`ReflectionMatrices(W)`

`CoreflectionMatrices(W)`

`Basis` MONSTGELT *Default :* "*Standard*"

The sequence of reflections in the finite real reflection group W. The rth reflection in the sequence corresponds to the rth (co)root.

`SimpleReflectionMatrices(W)`

`SimpleCoreflectionMatrices(W)`

`Basis` MONSTGELT *Default :* "*Standard*"

The matrices giving the action of the simple (co)roots on the (co)root space of the finite real reflection group W.

`ReflectionMatrix(W, r)`

`CoreflectionMatrix(W, r)`

`Basis` MONSTGELT *Default :* "*Standard*"

The reflection in finite real reflection group W corresponding to the rth (co)root. If $r = 1, \ldots, n$, this is a generator of W.

`SimpleReflectionPermutations(W)`

The sequence of permutations giving the action of the simple (co)roots of the finite reflection group W on the (co)roots. This action is the same for roots and coroots.

`ReflectionPermutations(W)`

The sequence of permutations giving the action of the (co)roots of the finite reflection group W on the (co)roots. This action is the same for roots and coroots.

`ReflectionPermutation(W, r)`

The permutation giving the action of the rth (co)root of the finite reflection group W on the (co)roots. This action is the same for roots and coroots.

`ReflectionWords(W)`

The sequence of words in the simple reflections for all the reflections of the real reflection group W. These words are given as sequences of integers. In other words, if $a = [a_1, \ldots, a_l] =$ `ReflectionWords(W)[r]`, then $s_{\alpha_r} = s_{\alpha_{a_1}} \cdots s_{\alpha_{a_l}}$.

`ReflectionWord(W, r)`

The word in the simple reflections for the rth reflection of the real reflection group W. The word is given as a sequence of integers. In other words, if $a = [a_1, \ldots, a_l] =$ `ReflectionWord(W,r)`, then $s_{\alpha_r} = s_{\alpha_{a_1}} \cdots s_{\alpha_{a_l}}$.

Example H105E25

```
> Q := RationalField();
> W := ReflectionGroup("A3");
> mx := ReflectionMatrix(W, 4);
> perm := ReflectionPermutation(W, 4);
> RootPosition(W, Vector(Q, Eltseq(Root(W,2))) * mx) eq 2^perm;
true
> mx := CoreflectionMatrix(W, 4);
> CorootPosition(W, Coroot(W,2) * mx) eq 2^perm;
true
```

`Length(w)`

`CoxeterLength(w)`

The length of w as an element of the Coxeter group W, ie. the number of positive roots of W which become negative under the action of w.

105.8.3 Weights

`WeightLattice(W)`

`CoweightLattice(W)`

The (co)weight lattice of the real reflection group W. The roots and coroots of W must have integral components.

`FundamentalWeights(W)`

`FundamentalCoweights(W)`

`Basis` MonStgElt *Default : "Standard"*

The fundamental weights of the real reflection group W given as the rows of a matrix. The roots and coroots of W must have integral components.

Example H105E26

```
> W := ReflectionGroup("E6");
> WeightLattice(W);
Lattice of rank 6 and degree 6
Basis:
(4  3  5  6  4  2)
(3  6  6  9  6  3)
(5  6 10 12  8  4)
(6  9 12 18 12  6)
(4  6  8 12 10  5)
(2  3  4  6  5  4)
Basis Denominator: 3
```

```
> FundamentalWeights(W);
[ 4/3    1  5/3    2  4/3  2/3]
[   1    2    2    3    2    1]
[ 5/3    2 10/3    4  8/3  4/3]
[   2    3    4    6    4    2]
[ 4/3    2  8/3    4 10/3  5/3]
[ 2/3    1  4/3    2  5/3  4/3]
```

`IsDominant(R, v)`

`Basis` MonStgElt *Default :* "*Standard*"

Returns `true` if and only if v is a dominant weight for the root datum R, ie, a nonnegative integral linear combination of the fundamental weights.

`DominantWeight(W, v)`

`Basis` MonStgElt *Default :* "*Standard*"

The unique dominant weight in the same W-orbit as v, where W is a real reflection group and v is a weight given as a vector or a sequence representing a vector. The second value returned is a Coxeter group element taking v to the dominant weight.

`WeightOrbit(W, v)`

`Basis` MonStgElt *Default :* "*Standard*"

The W-orbit of v as an indexed set, where W is a real reflection group and v is a weight given as a vector or a sequence representing a vector. The first element in the orbit is always dominant. The second value returned is a sequence of Coxeter group elements taking the dominant weight to the corresponding element of the orbit.

Example H105E27

```
> W := CoxeterGroup("B3");
> DominantWeight(W, [1,-1,0] : Basis:="Weight");
(1 0 0)
$.2 * $.3 * $.2 * $.1
> #WeightOrbit(W, [1,-1,0] : Basis:="Weight");
6
```

105.9 Related Structures

In this section we briefly list functions for creating other structures from a reflection group. See the appropriate chapters of the Handbook for more details.

`CoxeterGroup(GrpFPCox, W)`

The Coxeter group isomorphic to the real reflection group W. See Chapter 104.

`CoxeterGroup(GrpPermCox, W)`

The permutation Coxeter group isomorphic to the finite real reflection group W. See Chapter 104.

`LieAlgebra(W, R)`

The reductive Lie algebra over the ring R with Weyl group W. Unless W is finite, real, and crystallographic, an error is flagged. See Section 106.5.1.

`GroupOfLieType(W, k)`

The group of Lie type over the field k with Weyl group W. Unless W is finite, real, and crystallographic, an error is flagged. The roots and coroots of W must also have integral components. See Chapter 109.

105.10 Bibliography

[**Bou68**] N. Bourbaki. *Éléments de mathématique. Fasc. XXXIV. Groupes et algèbres de Lie. Chapitre IV: Groupes de Coxeter et systèmes de Tits. Chapitre V: Groupes engendrés par des réflexions. Chapitre VI: Systèmes de racines.* Hermann, Paris, 1968.

[**Bro10**] M. Broué. *Introduction to Complex Reflection Groups and their Braid Groups*, volume 1988 of *Lecture Notes in Mathematics*. Springer-Verlag, Berlin, 2010.

[**Car72**] Roger W. Carter. *Simple groups of Lie type*. John Wiley & Sons, London-New York-Sydney, 1972. Pure and Applied Mathematics, Vol. 28.

[**Coh76**] Arjeh M. Cohen. Finite complex reflection groups. *Ann. Sci. École Norm. Sup. (4)*, 9(3):379–436, 1976.

[**Hum90**] James E. Humphreys. *Reflection groups and Coxeter groups*, volume 29 of *Cambridge Studies in Advanced Mathematics*. Cambridge University Press, Cambridge, 1990.

[**Kan01**] Richard Kane. *Reflection Groups and Invariant Theory*, volume 5 of *CMS Books in Mathematics/Ouvrages de Mathématiques de la SMC*. Springer-Verlag, New York, 2001.

[**LT09**] G. I. Lehrer and D. E. Taylor. *Unitary Reflection Groups*, volume 20 of *Australian Mathematical Society Lecture Series*. Cambridge University Press, Cambridge, 2009.

[**ST54**] G. C. Shephard and J. A. Todd. Finite unitary reflection groups. *Canadian J. Math.*, 6:274–304, 1954.

106 LIE ALGEBRAS

106.1 Introduction 3213

106.1.1 Guide for the Reader 3213

106.2 Constructors for Lie Algebras 3214

`LieAlgebra< >` 3214
`LieAlgebra< >` 3214
`LieAlgebra< >` 3215
`LieAlgebra< >` 3215
`LieAlgebra(A)` 3215
`LieAlgebra(A)` 3215
`AbelianLieAlgebra(R, n)` 3216
`ChangeBasis(L, B)` 3216
`MatrixLieAlgebra(R, n)` 3216
`MatrixLieAlgebra(A)` 3217
`Algebra(M)` 3217
`LieAlgebra(M)` 3217

106.3 Finitely Presented Lie Algebras 3217

106.3.1 Construction of the Free Lie Algebra 3218

`FreeLieAlgebra(F, n)` 3218

106.3.2 Properties of the Free Lie Algebra 3218

`Rank(L)` 3218
`CoefficientRing(L)` 3218
`BaseRing(L)` 3218

106.3.3 Operations on Elements of the Free Lie Algebra 3219

`+ - *` 3219
`!` 3219
`Zero(L)` 3219
`IsLeaf(m)` 3219

106.3.4 Construction of a Finitely-Presented Lie Algebra 3220

`LieAlgebra(R)` 3220
`quo< >` 3223
`NilpotentQuotient(R, d)` 3223

106.3.5 Homomorphisms of the Free Lie Algebra 3224

`hom< >` 3224

106.4 Lie Algebras Generated by Extremal Elements 3225

106.4.1 Constructing Lie Algebras Generated by Extremal Elements . . . 3226

`ExtremalLieAlgebra(K, n)` 3226
`ExtremalLieAlgebra(K, G)` 3226

106.4.2 Properties of Lie Algebras Generated by Extremal Elements . . . 3227

`NumberOfGenerators(L)` 3227
`CoefficientRing(L)` 3227
`BaseRing(L)` 3227
`CommutatorGraph(L)` 3227
`Basis(L)` 3228
`ZBasis(L)` 3228
`Dimension(L)` 3228
`MultiplicationTable(~L)` 3229
`MultiplicationTable(L)` 3230

106.4.3 Instances of Lie Algebras Generated by Extremal Elements . . . 3231

`Instance(L)` 3231
`Instance(L, Q)` 3231

106.4.4 Studying the Parameter Space . 3233

`FreefValues(L)` 3233
`fValue(L, x, b)` 3233
`fValueProof(L, x, b)` 3233
`DimensionsEstimate(L, g)` 3234
`InstancesForDimensions(L, g, D)` 3235

106.5 Families of Lie Algebras . . 3236

106.5.1 Almost Reductive Lie Algebras . 3236

`LieAlgebra(T, k)` 3236
`MatrixLieAlgebra(T, k)` 3236
`LieAlgebra(N, k, p)` 3238
`LieAlgebra(R, k, p)` 3238
`TwistedLieAlgebra(R, k)` 3238

106.5.2 Cartan-Type Lie Algebras . . . 3239

`WittLieAlgebra(F, m, n)` 3241
`SpecialLieAlgebra(F, m, n)` 3241
`ConformalSpecialLieAlgebra(F, m, n)` 3241
`HamiltonianLieAlgebra(F, m, n)` 3242
`ConformalHamiltonianLieAlgebra(F, m, n)` 3242
`ContactLieAlgebra(F, m, n)` 3243

106.5.3 Melikian Lie Algebras 3244

`MelikianLieAlgebra(F, n1, n2)` 3244

106.6 Construction of Elements . . 3245

`Zero(L)` 3245
`!` 3245
`Random(L)` 3245

106.6.1 Construction of Elements of Structure Constant Algebras 3246

`elt< >` 3246
`!` 3246
`BasisProduct(L, i, j)` 3246
`BasisProducts(L)` 3246

106.6.2 Construction of Matrix Elements 3246

`elt< >` 3246
`!` 3246
`DiagonalMatrix(L, Q)` 3246
`ScalarMatrix(L, r)` 3246

106.7 Construction of Subalgebras, Ideals and Quotients 3247
sub< > 3247
ideal< > 3247
quo< > 3247
/ 3247
QuotientWithPullback(L, I) 3248

106.8 Operations on Lie Algebras . 3249
eq 3249
ne 3249
subset 3249
notsubset 3249
meet 3249
* 3250
^ 3250
Morphism(L, M) 3250
IsIsomorphic(L, M) 3250
IsKnownIsomorphic(L, M) 3250
IsIsomorphism(m) 3250
106.8.1 Basic Invariants 3252
CoefficientRing(L) 3252
BaseRing(L) 3252
Dimension(L) 3252
3252
Moduli(L) 3252
106.8.2 Changing Base Rings 3253
ChangeRing(L, S) 3253
ChangeRing(L, S, f) 3253
106.8.3 Bases 3253
BasisElement(A, i) 3253
. 3253
Basis(A) 3253
IsIndependent(Q) 3253
ExtendBasis(S, L) 3253
ExtendBasis(Q, L) 3253
106.8.4 Operations for Semisimple and Reductive Lie Algebras 3254
SemisimpleType(L) 3254
CartanName(L) 3254
ReductiveType(L) 3254
ReductiveType(L, H) 3254
RootSystem(L) 3256
RootDatum(L) 3257
ChevalleyBasis(L) 3257
ChevalleyBasis(L, H) 3257
ChevalleyBasis(L, H, R) 3258
IsChevalleyBasis(L, R, x, y, h) 3258
TwistedBasis(L, H, R) 3259
WeylGroup(L) 3260
WeylGroup(GrpPermCox, L) 3260
WeylGroup(GrpFPCox, L) 3261
WeylGroup(GrpMat, L) 3261

106.9 Operations on Subalgebras and Ideals 3261
DirectSum(L, M) 3261
IndecomposableSummands(L) 3261
DirectSumDecomposition(L) 3261
106.9.1 Standard Ideals and Subalgebras 3262
Centre(L) 3262
Center(L) 3262
Centraliser(L, K) 3262
Centralizer(L, K) 3262
Centraliser(L, x) 3262
Centralizer(L, x) 3262
Normaliser(L, K) 3262
Normalizer(L, K) 3262
SolubleRadical(L) 3262
SolvableRadical(L) 3262
Nilradical(L) 3262
106.9.2 Cartan and Toral Subalgebras . 3263
CartanSubalgebra(L) 3263
IsCartanSubalgebra(L, H) 3263
SplittingCartanSubalgebra(L) 3264
SplitMaximalToralSubalgebra(L) 3264
IsSplittingCartanSubalgebra(L, H) 3264
SplitToralSubalgebra(L) 3264
IsSplitToralSubalgebra(L, H) 3264
106.9.3 Standard Series 3265
CompositionSeries(L) 3265
CompositionFactors(L) 3265
MinimalIdeals(L : -) 3265
MaximalIdeals(L : -) 3266
DerivedSeries(L) 3266
LowerCentralSeries(L) 3266
UpperCentralSeries(L) 3266
106.9.4 The Lie Algebra of Derivations . 3267
LieAlgebraOfDerivations(L) 3267

106.10 Properties of Lie Algebras and Ideals 3268
KillingMatrix(L) 3268
IsAbelian(L) 3269
IsSoluble(L) 3269
IsSolvable(L) 3269
IsNilpotent(L) 3269
IsCentral(L, M) 3269
IsSimple(L) 3269
IsSemisimple(L) 3269
IsReductive(L) 3269
HasLeviSubalgebra(L) 3269
IsClassicalType(L) 3269

106.11 Operations on Elements . . 3270
+ - * 3270
IsCentral(L, M) 3270
NonNilpotentElement(L) 3270
AdjointMatrix(L, x) 3271
RightAdjointMatrix(L, x) 3271
106.11.1 Indexing 3271
a[i] 3271
a[i] := r 3271

a[i, j] 3272
a[i, j] := r 3272

106.12 The Natural Module 3272

Module(L) 3272
RModule(L) 3272
BaseModule(L) 3272
Degree(L) 3272
Degree(a) 3272
ElementToSequence(a) 3272
Eltseq(a) 3272
Coordinates(M, a) 3272
InnerProduct(a, b) 3272
Support(a) 3272

106.13 Operations for Matrix Lie Algebras 3273

BaseModule(M) 3273
Generic(M) 3273
Kernel(X) 3273
Nullspace(X) 3273
NullspaceOfTranspose(X) 3273
RowNullSpace(X) 3273

106.14 Homomorphisms 3273

hom< > 3273

106.15 Automorphisms of Classical-type Reductive Algebras . . 3274

IdentityAutomorphism(L) 3274
InnerAutomorphism(L, x) 3274
InnerAutomorphismGroup(L) 3274
DiagonalAutomorphism(L, v) 3274
GraphAutomorphism(L, p) 3274
DiagramAutomorphism(L, p) 3274

106.16 Restrictable Lie Algebras . . 3275

IsRestrictable(L) 3275
IsRestricted(L) 3275
IspLieAlgebra(L) 3275
RestrictionMap(L) 3275
pMap(L) 3275
RestrictedSubalgebra(Q) 3275
pSubalgebra(Q) 3275
pClosure(L, M) 3276
IsRestrictedSubalgebra(L, M) 3276
IspSubalgebra(L, M) 3276
pQuotient(L, M) 3276
JenningsLieAlgebra(G) 3276

106.17 Universal Enveloping Algebras 3277

106.17.1 Background 3277

106.17.2 Construction of Universal Enveloping Algebras 3278

UniversalEnvelopingAlgebra(L) 3278
IntegralUEA(L) 3278
IntegralUEAlgebra(L) 3278
IntegralUniversalEnvelopingAlgebra(L) 3278
AssignNames(~U, Q) 3279
ChangeRing(U, S) 3279

106.17.3 Related Structures 3279

CoefficientRing(U) 3279
BaseRing(U) 3279
Algebra(U) 3279

106.17.4 Elements of Universal Enveloping Algebras 3279

! 3280
Zero(U) 3280
! 3280
One(U) 3280
. 3280
! 3280
HBinomial(U, i, n) 3280
HBinomial(h, n) 3280
+ - * * * ^ 3281
Monomials(u) 3281
Coefficients(u) 3281
Degree(u, i) 3281

106.18 Solvable and Nilpotent Lie Algebras Classification 3282

106.18.1 The List of Solvable Lie Algebras 3282

106.18.2 Comments on the Classification over Finite Fields 3283

106.18.3 The List of Nilpotent Lie Algebras 3284

106.18.4 Intrinsics for Working with the Classifications 3285

SolvableLieAlgebra(F, n, k : -) 3285
NilpotentLieAlgebra(F, r, k : -) 3286
AllSolvableLieAlgebras(F, d) 3286
AllNilpotentLieAlgebras(F, d) 3286
IdDataSLAC(L) 3286
IdDataNLAC(L) 3287
MatrixOfIsomorphism(f) 3287

106.19 Semisimple Subalgebras of Simple Lie Algebras 3289

SubalgebrasInclusionGraph(t) 3289
RestrictionMatrix(G, k) 3290

106.20 Nilpotent Orbits in Simple Lie Algebras 3291

IsGenuineWeighted DynkinDiagram(L, wd) 3292
NilpotentOrbit(L, wd) 3292
NilpotentOrbit(L, e) 3292
NilpotentOrbits(L) 3293
Partition(o) 3293
SL2Triple(o) 3294
SL2Triple(L, e) 3294
Representative(o) 3294
WeightedDynkinDiagram(o) 3294

106.21 Bibliography 3295

Chapter 106
LIE ALGEBRAS

106.1 Introduction

This chapter is concerned with finite dimensional Lie algebras. A large number of specialised functions are provided for these algebras. We refer to [dG00] for a general introduction to the theory of Lie algebras and their algorithms. For some of the functions described here that rely on a non-trivial algorithm we will indicate a precise reference.

Lie algebras are viewed as free modules over a base ring R with a multiplication satisfying the usual Lie axioms. Some functions require additional conditions on the base ring; for example, many functions require that the base ring be a field.

The main computational machinery in MAGMA for Lie algebras assumes that they are given either as structure constant algebras or as matrix algebras. Functions are provided which, given a finitely presented finite dimensional Lie algebra, will attempt to construct an isomorphic structure constant Lie algebra. As a structure constant algebra, the Lie algebra L of dimension n over a ring R is defined in MAGMA by giving the n^3 *structure constants* $a_{ij}^k \in R (1 \leq i, j, k \leq n)$ such that, if $\{e_1, e_2, \ldots, e_n\}$ is the basis of L, $e_i * e_j = \sum_{k=1}^{n} a_{ij}^k * e_k$.

In MAGMA V2.22 there is more functionality for Lie algebras defined by structure constants than for matrix Lie algebras. Throughout this chapter the algebra representation appropriate for a given intrinsic will be noted. For information on matrix algebras considered as associative algebras see Chapter 88.

In addition, some functions are provided for finitely presented Lie algebras and Lie algebras generated by extremal elements, and databases of solvable Lie algebras, nilpotent Lie algebras, and nilpotent orbits in simple Lie algebras are available.

106.1.1 Guide for the Reader

As mentioned above, the most extensively supported Lie algebras in MAGMA are structure constant Lie algebras and matrix Lie algebras. The methods for constructing these (by explicitly specifying structure constants or matrices) are described in Sections 106.2.

Well known simple finite Lie algebras can be more easily constructed by specifying the type. The classical, reductive, Lie algebras (A_n, B_n, C_n, D_n, E_6, E_7, E_8, F_4, G_2) and their twisted variants are described in Section 106.5.1, the Witt Lie algebras and its derivatives (of *Cartan-Type*) in Section 106.5.2, and the Melikian Lie algebras in Section 106.5.3. Those interested primarily in classical Lie algebras may want to skip to Section 106.5.1, which includes a number of examples to get started.

Elementary properties of these Lie algebras (bases, types, Weyl group, etc.) are described in Section 106.8. This section also contains information about isomorphism testing. Other properties (nilpotency, simplicity) are discussed in Section 106.10. Some further operations that only apply to matrix Lie algebras may be found in Section 106.13.

Constructors such as direct sums, subalgebras, centralisers, Cartan subalgebras, derived series, etc. are treated in Sections 106.7 and 106.9; construction of homomorphisms can be found in Section 106.14. The construction of elements of Lie algebras is described in Section 106.6, and operations on them in Section 106.11.

In addition to these Lie algebras MAGMA supports two other types of Lie algebras. Firstly finitely presented Lie algebras, as free Lie algebras modulo a set of relations, described in Section 106.3. Secondly Lie algebras generated by extremal elements, described in Section 106.4.

More specialistic functions for structure constant Lie algebras are described in Sections 106.12 (the natural module), 106.15 (automorphisms of classical-type Lie algebras), 106.16 (restrictable Lie algebras), and 106.17 (universal enveloping algebras).

MAGMA also provides databases and recognition procedures for small-dimensional solvable and nilpotent Lie algebras (described in Section 106.18), a database of semisimple subalgebras of simple Lie algebras (described in Section 106.19), and a database of nilpotent orbits in simple Lie algebras (described in Section 106.20).

Other chapters that may be of interest are Chapter 107 on Kac-Moody Lie algebras and Chapter 108 on Quantum Groups. Furthermore, Chapter 110 deals with representations of Lie algebras and groups of Lie type. Of particular importance is Section 110.3.1, dealing with the construction of representations of Lie algebras.

106.2 Constructors for Lie Algebras

The construction of a Lie algebra defined by structure constants is identical to that of a general structure constant algebra. Most constructors take two optional parameters: `Check` and `Rep`.

By default, the conditions for the algebra to be a Lie algebra are checked. If the user decides to omit this check, by setting the parameter `Check` to `false`, and the algebra is not actually Lie then functions in this section will fail or give incorrect answers.

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

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

`LieAlgebra< M | Q : parameters >`

`Check`	BOOLELT	*Default :* `true`
`Rep`	MONSTGELT	*Default :* "*Dense*"

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

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

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

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

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

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

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

`LieAlgebra< t | T :` *parameters* `>`

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

This function creates the Lie structure constant algebra L over the integers, with standard basis $\{e_1, e_2, \ldots, e_n\}$. The sequence T contains quadruples $< i, j, k, a_{ij}^k >$ (where the a_{ij}^k are integers) giving the non-zero structure constants. All other structure constants are defined to be 0. The argument t is a sequence of length n consisting of nonnegative integers giving the moduli of the basis elements. Thus let t_i denote the i-th element of t; then $t_i e_i = 0$. So if $t_i = 0$, then $ke_i \neq 0$ for all integers k.

`LieAlgebra(A)`

Given an associative structure-constant algebra A, create the Lie algebra L consisting of the elements in A with the induced Lie product $(x, y) \to x * y - y * x$. As a second value the map identifying the elements of L and A is returned.

`LieAlgebra(A)`

Given an associative matrix algebra A, create a structure-constant Lie algebra L isomorphic to A with the induced Lie product $(x, y) \to x * y - y * x$.

`AbelianLieAlgebra(R, n)`

`Rep` MonStgElt *Default :* "*Sparse*"

Create the abelian Lie algebra of dimension n over the ring R.

Example H106E1

We construct the Heisenberg Lie algebra, then a Lie algebra from an associative algebra, and finally a Lie algebra over the integers (also called a Lie ring).

```
> T:= [ <1,2,3,1>, <2,1,3,-1> ];
> LieAlgebra< Rationals(), 3 | T >;
Lie Algebra of dimension 3 with base ring Rational Field
> A:= Algebra( GF(27), GF(3) );
> LieAlgebra(A);
Lie Algebra of dimension 3 with base ring GF(3)
> T:= [ <1,2,2,2>, <2,1,2,2> ];
> t:= [0,4];
> K:= LieAlgebra< t | T : Rep:= "Dense" >; K;
Lie Algebra of dimension 2 with base ring Integer Ring
Column moduli: [0, 4]
> LowerCentralSeries( K );
[
    Lie Algebra of dimension 2 with base ring Integer Ring
    Column moduli: [0, 4],
    Lie Algebra of dimension 1 with base ring Integer Ring
    Column moduli: [2],
    Lie Algebra of dimension 0 with base ring Integer Ring
]
```

`ChangeBasis(L, B)`

`Rep` MonStgElt *Default :* "*Dense*"

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

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

`MatrixLieAlgebra(R, n)`

Given a ring R and an integer n, create the full Lie algebra of matrices of degree d over R.

`MatrixLieAlgebra(A)`

Given an associative matrix algebra A, create the matrix Lie algebra L consisting of the elements in A with the induced Lie product $(x, y) \to x * y - y * x$.

`Algebra(M)`

`LieAlgebra(M)`

Return a structure-constant Lie algebra isomorphic to the matrix Lie algebra M.

Example H106E2

We construct the subalgebra of the matrix Lie algebra of 2×2 matrices, consisting of upper triangular matrices.

```
> L:= MatrixLieAlgebra( Rationals(), 2 );
> a:= L!Matrix( [[1,0],[0,0]] );
> b:= L!Matrix( [[0,0],[1,0]] );
> c:= L!Matrix( [[0,0],[0,1]] );
> K:= sub< L | [ a, b, c ] >;
> Dimension(K);
3
> IsSolvable(K);
true
> IsNilpotent(K);
false
```

106.3 Finitely Presented Lie Algebras

A finitely presented Lie algebra is constructed as the quotient of a free Lie algebra on a finite number of generators. Denote the set of generators by $X = \{x_1, \ldots, x_n\}$. Let F denote the base ring. Then the free Lie algebra generated by the x_i over the ring F is denoted by $L_F(X)$. The free magma on X is the set of the x_i together with all bracketed expressions in the x_i, e.g., $((x_1, x_2), ((x_1, x_3), x_2)))$. The free Lie algebra $L_F(X)$ is spanned by $M(X)$. However, the elements of this set are not linearly independent. It is a nontrivial problem to describe a basis of the free Lie algebra. One of several possibilities is the well-known *Hall basis*. Currently MAGMA does not support calculations involving bases of the free Lie algebra, as they are of little use for our main problem: the construction of a basis and multiplication table for a finitely-presented Lie algebra.

It is convenient to define an ordering on the elements of $M(X)$. First of all, each generator is assigned a degree. Usually, the degree of all x_i is taken to be one, but it is also possible to assign different degrees. The degree of a bracket (a, b) is defined to be the sum of the degrees of a and b. Let m, m' be two elements of $M(X)$. Then define $m < m'$ if the degree of m is less than the degree of m'. If their degrees are equal, then define $m < m'$ if $m = x_i$ and $m' = (a', b')$, for some a', b' in $M(X)$. If both m and m' are generators of the same degree, so that $m = x_i$, $m' = x_j$, then define $m < m'$ if $i < j$. Finally, if both m and

m' are bracketed expressions, that is, $m = (a, b)$ and $m' = (a', b')$, then define $m < m'$ if $a < a'$ or $a = a'$ and $b < b'$.

In the free Lie algebra, the relations $(a, b) = -(b, a)$, and $(a, a) = 0$ hold. In MAGMA this is used to rewrite an arbitrary element as a linear combination of elements of the form (a, b) with $a < b$. If instead we were to work relative to a basis for $L_F(X)$, then the use of the Jacobi identity when rewriting elements can lead to rather large expressions. Thus, mathematically speaking, in MAGMA rather than work in the free Lie algebra, we actually work in the free nonassociative anticommutative algebra. However, as our main interest lies in finitely-presented Lie algebras, this is usually not a problem.

106.3.1 Construction of the Free Lie Algebra

`FreeLieAlgebra(F, n)`

Given a ring F and a positive integer n, this function creates the free n-generator Lie algebra over the ring F. The generators are ordered, with the first generator being the biggest in the ordering, and the last generator the smallest. The angle bracket notation can be used to assign names to the generators.

Example H106E3

The following statement creates the MAGMA object corresponding to the free Lie algebra on three generators over the field $\mathbf{F}_2$.

```
> L<a,b,c>:= FreeLieAlgebra(GF(2), 3);
```

106.3.2 Properties of the Free Lie Algebra

`Rank(L)`

The number of generators of the free Lie algebra L.

`CoefficientRing(L)`

`BaseRing(L)`

The coefficient ring of L.

106.3.3 Operations on Elements of the Free Lie Algebra

Once a free Lie algebra has been created the user can construct a bracketed expression (a, b), either by simply typing it literally as `(a,b)`, or by using the multiplication operator as in `a*b`. Recall that MAGMA rewrites elements so they are in the form (a, b) with $a < b$. On some occasions this can lead to the introduction of a minus sign. Also, if an element contains a subexpression of the form (a, a), it will be rewritten to 0.

We can multiply and add elements, and multiply them by scalars.

```
x + y
```

```
x - y
```

```
x * y
```

```
L ! 0
```

```
Zero(L)
```

The zero element of the free Lie algebra L.

Example H106E4

```
> L<z,y,x> := FreeLieAlgebra(Rationals(), 3);
> x*y;
(x, y)
> (x, y);
(x, y)
> ((x*y)*z);
-(z, (x, y))
> ((x, y), z);
-(z, (x, y))
> ((x, y),(y, x));
0
> 2*((x, y), z) - ((x, z), (y, z)) + 1/2*(x, (x, (y, z)));
-((x, z), (y, z)) + 1/2*(x, (x, (y, z))) - 2*(z, (x, y))
```

```
IsLeaf(m)
```

Given a monomial element m of the free Lie algebra L, return `true` if m is a generator and `false` otherwise. If the result is `true` then the second return value is an integer i such that m is `L.i`. If the result is `false` then $a, b \in L$ are also returned such that m is a multiple of (a, b).

Note that in the latter case m is not equal to (a, b), but merely equal to a scalar multiple of (a, b). See the example for a possible method of retrieving the appropriate scalar.

Example H106E5

```
> L<z,y,x>:= FreeLieAlgebra(Rationals(), 3);
> IsLeaf(x);
true 3
> m := 2*((x, y), z);
> m;
-2*(z, (x, y))
> il, a, b := IsLeaf(m);
> il, a, b;
false z
(x, y)
> m eq (a, b);
false
> m eq LeadingCoefficient(m)*(a,b);
true
```

106.3.4 Construction of a Finitely-Presented Lie Algebra

`LieAlgebra(R)`

Given a set or sequence R of elements of a free Lie algebra L, let I be the ideal of L generated by the elements of R. It is assumed that the quotient algebra $Q = L/I$ is finite dimensional. This function returns the structure constant Lie algebra K isomorphic to the quotient Q. If the quotient Q is infinite dimensional then the program will not terminate. (The question of determining whether the quotient is finite dimensional is known to be undecidable.) The function can be interrupted by pressing `Ctrl-C`. The elements of R are referred to as *relations*.

This function works if the base ring is either a field or equal to the ring of integers. In these two cases slightly different objects are returned.

If the base ring is a field then four values are returned:

(a) A structure constant algebra K isomorphic to the quotient Q;

(b) A sequence G comprising sequences of integers;

(c) A sequence B of elements of the free Lie algebra L;

(d) A map $f : B \times B \to L$.

The sequence B maps to a basis of the quotient algebra, so it is a basis of a complement of the ideal I in the free Lie algebra L. The elements of B are in one-to-one correspondence with the basis elements of K.

If all the relations of R are homogeneous (i.e., if they are linear combinations of elements of the same degree), then Q is graded. The sequence G contains information about the grading. It consists of sequences of length two. The first element of each subsequence is the degree of a homogeneous subspace H, while the second element is the dimension of H. The basis elements of K are ordered with respect to increasing

degree. So from G it is straightforward to read off the degree of each basis element. If the relations are not homogeneous then the sequence G is empty. Finally, f is a map that takes two elements from B as arguments, and returns their product (in L) modulo the ideal I. The algorithm used is described in [dG00], §7.4.

Secondly, in the case in which the base ring is the ring of integers, four values are returned:

(a) A structure constant algebra K isomorphic to the quotient Q;

(b) A sequence G comprising sequences of integers;

(c) A sequence B that is always empty;

(d) A map $f : K \to L$.

Here the structure constant algebra is defined over the ring of integers, so it may have torsion. The sequence G is nonempty only if the input relations are homogeneous in which case it contains the dimensions of the homogeneous components. The function f is a map that takes an element u of K and returns an element of the free algebra L that maps to u under the projection map (from the free algebra to the quotient).

Example H106E6

In this example we compute the subalgebra K of E_7 spanned by the positive root spaces.

```
> L<x7,x6,x5,x4,x3,x2,x1>:= FreeLieAlgebra(RationalField(), 7);
> pp:= { [1,3], [3,4], [2,4], [4,5], [5,6], [6,7] };
> R:= [ ];
> g:= [ L.i : i in [1..7] ];
> for i in [1..7] do
>     for j in [i+1..7] do
>         if [i,j] in pp then
>            a:= (g[i],(g[i],g[j]));
>            Append( ~R, a );
>            Append( ~R, (g[j],(g[j],g[i])) );
>         else
>            Append( ~R, (g[i],g[j]) );
>         end if;
>     end for;
> end for;
> R;
[
    -(x6, x7), -(x7, (x5, x7)), (x5, (x5, x7)), -(x4, x7),
    -(x3, x7), -(x2, x7), -(x1, x7), -(x5, x6), -(x6, (x4, x6)),
    (x4, (x4, x6)), -(x3, x6), -(x2, x6), -(x1, x6), -(x5, (x4, x5)),
    (x4, (x4, x5)), -(x3, x5), -(x2, x5), -(x1, x5), -(x4, (x3, x4)),
    (x3, (x3, x4)), -(x2, x4), -(x1, x4), -(x3, (x2, x3)), (x2, (x2, x3)),
    -(x1, x3), -(x2, (x1, x2)), (x1, (x1, x2))
]
> time K, G, B, f := LieAlgebra(R);
```

```
Time: 0.280
> K;
Lie Algebra of dimension 63 with base ring Rational Field
> #B;
63
> B[63];
(x7, (x5, (x4, (x3, (x2, (x1, (x6, (x4, (x3, (x2, (x5, (x4,
    (x3, (x6, (x4, (x5, x7))))))))))))))))
```

Example H106E7

In this example we construct a finitely presented Lie ring (i.e., Lie algebra over the integers).

```
> L<y,x>:= FreeLieAlgebra( Integers(), 2 );
> R:= [ x*(x*(x*y))-2*x*y, 2*y*(x*(x*y)), 3*y*(y*(x*y))-x*(x*y),
>  x*(y*(x*(y*(x*y)))) ];
> K,g,b,f:= LieAlgebra( R );
> K;
Lie Algebra of dimension 8 with base ring Integer Ring
Column moduli: [2, 2, 2, 8, 8, 8, 0, 0]
> f(K.4);
(y, (x, y))
> LowerCentralSeries( K );
[
    Lie Algebra of dimension 8 with base ring Integer Ring
    Column moduli: [2, 2, 2, 8, 8, 8, 0, 0],
    Lie Algebra of dimension 6 with base ring Integer Ring
    Column moduli: [2, 2, 2, 8, 8, 8],
    Lie Algebra of dimension 6 with base ring Integer Ring
    Column moduli: [2, 2, 2, 4, 8, 8],
    Lie Algebra of dimension 6 with base ring Integer Ring
    Column moduli: [2, 2, 2, 4, 4, 8],
    Lie Algebra of dimension 6 with base ring Integer Ring
    Column moduli: [2, 2, 2, 4, 4, 4],
    Lie Algebra of dimension 5 with base ring Integer Ring
    Column moduli: [2, 2, 2, 4, 4],
    Lie Algebra of dimension 4 with base ring Integer Ring
    Column moduli: [2, 2, 2, 4],
    Lie Algebra of dimension 3 with base ring Integer Ring
    Column moduli: [2, 2, 2],
    Lie Algebra of dimension 2 with base ring Integer Ring
    Column moduli: [2, 2],
    Lie Algebra of dimension 1 with base ring Integer Ring
    Column moduli: [2],
    Lie Algebra of dimension 0 with base ring Integer Ring
]
```

`quo< L | R >`

This function is similar to the function `LieAlgebra` in that it constructs a structure constant Lie algebra K isomorphic to the quotient L/I, where I is the ideal of L generated by the elements (relations) of the sequence R. In addition to K, an invertible map from L to K is returned.

Example H106E8

In this example we demonstrate the use of the quotient constructor for finitely presented Lie algebras.

```
> L<x,y> := FreeLieAlgebra(Rationals(), 2);
> R := [ x*(x*y)-2*x, y*(y*x)-2*y ];
> K, phi := quo<L | R>;
> K;
Lie Algebra of dimension 3 with base ring Rational Field
> SemisimpleType(K);
A1
> [ b @@ phi : b in Basis(K) ];
[
   y,
   x,
   (y, x)
]
> phi(x*y);
( 0  0 -1)
```

`NilpotentQuotient(R, d)`

Given a set or sequence R of elements of a free Lie algebra L, let I be the ideal of L generated by the elements of R. Let d be a positive integer or `Infinity()`. This function constructs the class d nilpotent quotient of the Lie algebra L/I, a finite dimensional algebra. The function returns the same values as `LieAlgebra`.

This function is similar to the function `LieAlgebra` except that the quotient is constructed in the free nilpotent Lie algebra of class d. All elements of degree strictly larger than `d` will be added to the ideal, so the quotient will be finite-dimensional and nilpotent of class at most `d`.

Example H106E9

In this example, we compute a nilpotent quotient.

```
> L<y,x> := FreeLieAlgebra(Rationals(), 2);
> R := [(x, (x, (x, y))) - (y, (y, (x, y)))];
> time K, G, B, f := NilpotentQuotient(R, 10);
Time: 0.040
> K;
```

```
Lie Algebra of dimension 109 with base ring Rational Field
> #B;
109
> B[100];
(y, (x, (x, (y, (x, (y, (x, (x, (x, y)))))))))
> G;
[
    [ 1, 2 ],
    [ 2, 1 ],
    [ 3, 2 ],
    [ 4, 2 ],
    [ 5, 4 ],
    [ 6, 5 ],
    [ 7, 10 ],
    [ 8, 15 ],
    [ 9, 26 ],
    [ 10, 42 ]
]
> f(B[3], B[13]);
-(y, (x, (x, (x, (x, (y, (x, y)))))))) + (x, (y, (x, (x, (x, (y, (x, y)))))))
```

106.3.5 Homomorphisms of the Free Lie Algebra

`hom< L -> M | Q >`

Given a free Lie algebra L of dimension n over R and either a Lie algebra M over R or a module M over R, construct the homomorphism from L to M specified by Q. The sequence Q must have length `Rank(L)` and be of the form $[m_1, \ldots, m_n]$ ($m_i \in M$) indicating that the i-th generator of L maps to m_i.

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

Example H106E10

We construct the Lie algebra of type A_1 as quotient of a free Lie algebra, using homomorphisms between a free Lie algebra and a structure constant Lie algebra. First, we construct the free Lie algebra and a structure constant Lie algebra of type A_1. The elements of a Chevalley basis are obtained by a call to `ChevalleyBasis`.

```
> L<e,f> := FreeLieAlgebra(Rationals(), 2);
> M := LieAlgebra("A1", Rationals() : Isogeny := "SC");
> pos, neg, cart := ChevalleyBasis(M);
> pos, neg, cart;
[ (0 0 1) ]
[ (1 0 0) ]
```

```
[ (0 1 0) ]
```

Next, we construct a homomorphism from L to M that sends e to the positive root element, and f to the negative root element. We construct a map from M to L that sends the positive root to e, the negative root to f, and the Cartan element to $-(e, f)$.

```
> phi := hom<L -> M | [ pos[1], neg[1] ]>;
> phi(e), phi(f), phi(e*f);
(0 0 1) (1 0 0) ( 0 -1  0)
> imgs := [ L | f, (f,e), e];
> psi := map<M -> L | x :-> &+[ x[i]*imgs[i] : i in [1..3] ]>;
> psi(cart[1]);
-(f, e)
> psi(phi((e,(e,f))));
-2*e
> assert forall{b : b in Basis(M) | phi(psi(b)) eq b };
```

Finally, we create a sequence of relations showing that the maps `phi` and `psi` are each others inverses for a small set of elements of L. We then compute the quotient of the free Lie algebra with respect to these relations.

```
> R := { x - psi(phi(x)) : x in {e, f, (e,f), (e,(e,f)), (f,(f,e))} };
> L2 := quo<L | R>;
> L2;
Lie Algebra of dimension 3 with base ring Rational Field
> SemisimpleType(L2);
A1
```

106.4 Lie Algebras Generated by Extremal Elements

A non-zero element x of a Lie algebra L over the field K is extremal if $[x, [x, y]] \in Kx$ for all $y \in L$. If x is extremal, the existence of a linear map $f_x : L \to K$ such that $[x, [x, y]] = f_x(y)x$ for all $y \in L$ immediately follows from linearity of $[\cdot, \cdot]$.

In this section we describe functions for computing with Lie algebras generated by such extremal elements. For a simple connected undirected finite graph Γ we consider an algebraic variety X over K whose K-points parametrize Lie algebras generated by extremal elements. Here the generators of the Lie algebras correspond to the vertices of the graph, and we prescribe commutation relations corresponding to the nonedges of Γ.

Details of the setup may be found in [Roo11]; we describe the essential ingredients here.

Assume that Γ is a connected undirected finite graph with n vertices, without loops or multiple bonds, and that K is a field of characteristic distinct from 2. We let Π be the vertex set of Γ and denote adjacency of two vertices $x, y \in \Pi$ by $i \sim j$.

We denote by $F(K, \Gamma)$ the quotient of the free Lie algebra over K generated by Π modulo the relations $[x, y] = 0$ for all $x, y \in \Pi$ with $x \not\sim y$. We write F^* for the space of all K-linear functions on F. For every $f \in (F^*)^\Pi$ we denote by $L(K, \Gamma, f)$ (often abbreviated to

$L(f)$) the quotient of $F(K,\Gamma)$ by the ideal $I(f)$ generated by the infinitely many elements $[x,[x,y]] - f_x(y)x$ for $x \in \Pi$, $y \in F$.

By construction $L(f)$ is a Lie algebra generated by $|\Pi| = n$ extremal elements, the extremal generators corresponding to the vertices of Γ and commuting whenever they are not adjacent. The element $f_x \in F^*$ is a parameter expressing the extremality of $x \in \Pi$.

In the Lie algebra $L(0)$ the elements of Π map to sandwich elements. This algebra is finite-dimensional, by [ZK90] this Lie algebra is finite-dimensional; for general $f \in (F^*)^\Pi$ we have $\dim(L(f)) \leq \dim(L(0))$ by [CSUW01, Lemma 4.3]. It is therefore natural to focus on the Lie algebras $L(f)$ of maximal possible dimension, i.e., those of dimension $\dim(L(0))$. We define the set $X := \{f \in (F^*)^\Pi \mid \dim(L(f)) = \dim(L(0))\}$, the parameter space for all maximal-dimensional Lie algebras of the form $L(f)$.

The functions currently implemented in MAGMA allow computation of X and $L(f)$, for any f, for the cases where X is an affine space (which is unproven, but true in all currently known cases).

Lie algebras generated by extremal elements are of type `AlgLieExtr`. The verbose flag `"AlgLieExtr"` may be set between 1 and 5 to show details and progress of the various computations.

106.4.1 Constructing Lie Algebras Generated by Extremal Elements

`ExtremalLieAlgebra(K, n)`

`CommGens`	SEQENUM	*Default :* []
`HeisenbergPairs`	SEQENUM	*Default :* []

Construct the Lie algebra over the field K generated by n extremal elements. The characteristic of K must be distinct from 2.

The optional argument `CommGens` contains pairs of integers (i,j), with $1 \leq i,j \leq n$, describing that generators x_i and x_j commute, i.e., $[x_i,x_j] = 0$.

The optional argument `HeisenbergPairs` contains pairs of integers (i,j), with $1 \leq i \leq n$ and $1 \leq j \leq \dim(L(0))$, describing that $f_{x_i}(b_j)$ should be taken equal to 0. (Note that if it is required to have $j > n$ it would be necessary to have prior knowledge about the basis of $L(0)$).

`ExtremalLieAlgebra(K, G)`

`HeisenbergPairs`	SEQENUM	*Default :* []

Construct the Lie algebra over the field K whose extremal generators are described by the graph G, i.e., with $|V(G)|$ generators, and x_i and x_j commute whenever vertices x_i and x_j of G are not adjacent.

See `ExtremalLieAlgebra` above for a description of the optional argument `HeisenbergPairs`.

106.4.2 Properties of Lie Algebras Generated by Extremal Elements

`NumberOfGenerators(L)`

The number of generators of L.

`CoefficientRing(L)`

`BaseRing(L)`

The coefficient ring of L. Immediately after construction, this is equal to the field K provided as argument to `ExtremalLieAlgebra`. However, after the multiplication table has been computed (see below), the coefficient ring would in general be a multivariate polynomial ring over K describing the parameter space.

`CommutatorGraph(L)`

The graph describing the extremal generators of L and their commutator relations.

Example H106E11

We construct a Lie algebra generated by 4 extremal elements in two different manners.

```
> QQ := Rationals();
> L := ExtremalLieAlgebra(QQ, BipartiteGraph(2,2));
> Ngens(L), CoefficientRing(L);
4 Rational Field
> G := CommutatorGraph(L); G;
Graph
Vertex  Neighbours
1       3 4 ;
2       3 4 ;
3       1 2 ;
4       1 2 ;
> L := ExtremalLieAlgebra(QQ, 4 : CommGens := [<1,2>,<3,4>]);
> Ngens(L), CoefficientRing(L);
4 Rational Field
> G := CommutatorGraph(L); G;
Graph
Vertex  Neighbours
1       3 4 ;
2       3 4 ;
3       1 2 ;
4       1 2 ;
```

`Basis(L)`

Compute a monomial basis for $L(0)$ (this is also a monomial basis for $L(f)$ for any $f \in X$; see the introduction of Section 106.4).

The first return value is a sequence consisting of monomials of the free Lie algebra over K with n generators, where K is the coefficient ring of L and n is the number of generators. The second return value is a sequence consisting of functions c. Each of these functions may be applied to a sequence of generators and a composition function. These may be used to construct the basis elements in other environments.

The algorithm used in this function is due to W. de Graaf.

`ZBasis(L)`

For L a Lie algebra generated by extremal elements over the field of rational numbers, compute a basis of the corresponding Lie ring over the integers.

This function returns three sequences B, T, C, respectively, describing bases for $L(0)$ over **any** field K. B is a not necessarily monomial basis, with torsion described by T. It is such that if $T[i]$ is nonzero, m say, then $B[i]$ is zero unless the characteristic of K divides m.

The third sequence, C, is a sequence of monomials that linearly span $L(0)$ over any field K. Note, however, that if T contains nonzero elements, then C would in general contain superfluous elements and therefore not be a basis.

The algorithm used in this function is due to W. de Graaf. The only currently known case with nontrivial torsion is for $\Gamma(L) = K_5$.

`Dimension(L)`

The dimension of $L(0)$. This value is computed via a basis computation, so potentially quite time-consuming.

Example H106E12

We continue the previous example H106E11 and demonstrate the computation of a basis of $L(0)$.

```
> B, C := Basis(L);
> B;
[
    $.1,
    $.2,
    $.3,
    $.4,
    ($.4, $.2),
    ($.4, $.1),
    ($.3, $.2),
    ($.3, $.1),
    ($.4, ($.3, $.2)),
    ($.4, ($.3, $.1)),
    ($.2, ($.4, $.1)),
    ($.2, ($.3, $.1)),
```

```
    ($.4, ($.2, ($.3, $.1))),
    ($.3, ($.2, ($.4, $.1))),
    ($.2, ($.4, ($.3, $.1)))
]
> [ c(["x","y","z","u"], func<i,j|i cat j>) : c in C ];
[ x, y, z, u, uy, ux, zy, zx, uzy, uzx, yux, yzx, uyzx, zyux, yuzx ]
> A := FreeAlgebra(Rationals(), 4);
> [ c([A.1,A.2,A.3,A.4], func<x,y|x*y>) : c in C ];
[
    $.1,
    $.2,
    $.3,
    $.4,
    $.4*$.2,
    $.4*$.1,
    $.3*$.2,
    $.3*$.1,
    $.4*$.3*$.2,
    $.4*$.3*$.1,
    $.2*$.4*$.1,
    $.2*$.3*$.1,
    $.4*$.2*$.3*$.1,
    $.3*$.2*$.4*$.1,
    $.2*$.4*$.3*$.1
]
> #B, #C, Dimension(L);
15 15 15
```

`MultiplicationTable(~L)`

`HowMuch`	MonStgElt	*Default :* "*Auto*"
`MemLimit`	RngIntElt	*Default :* ∞
`FullJacobi`	BoolElt	*Default :* `false`

Force computation of a general multiplication table for L, i.e., one that may be used for constructing $L(f)$ for any $f \in X$ (see the introduction to this section 106.4). This computation is necessary for constructing instances as described in Section 106.4.3, but it will be done automatically if needed. Data about the variety X is computed concurrently and stored internally; see Section 106.4.4 for the relevant functions in accessing that information.

The optional parameters may be used to influence the computation, although the defaults should generally work well. `HowMuch` may be set to "Auto" (the default), "Top" or "Full" and prescribes whether only the first `Ngens(L)` rows of the multiplication table are computed ("Top"), or all entries ("Full"). If set to "Auto" some fraction of the multiplication table is computed depending on the dimension of L and the other parameters.

`MemLimit` may be set to a positive integer m, and if given MAGMA will attempt to limit its memory usage to m MB, by limiting the portion of the multiplication table that is being computed.

`FullJacobi` may be set to `true` in order to force checking the Jacobi identity for all basis elements, thus providing more certainty with regards to the information about the parameter space X. Note that even if this parameter is set to `true` a heuristic (Monte-Carlo) method is used, as considering all $\dim(L(0))^3$ triples quickly becomes infeasible as the dimension grows.

The verbose flag `"AlgLieExtr"` may be set to 3 or more to obtain some information about the default choices MAGMA makes with regards to these parameters.

`MultiplicationTable(L)`

`Rep`	MONSTGELT	*Default :* "*Auto*"
`Check`	BOOLELT	*Default :* `true`

A general multiplication table for L.

If `Rep` is set to "Dense" it will be returned as a sequence of sequences of vectors over `CoefficientRing(L)`. If `Rep` is set to "Sparse" it will be returned as a sequence of 4-tuples. If `Rep` is set to "Auto" a choice between these representations is made depending on $\dim(L)$. Both these representations may be used on the right hand side of the `LieAlgebra` constructor.

The optional parameter `Check` controls whether the Jacobi identity is verified for all triples (if `true` it will actually be checked for all $\dim(L(0))^3$ triples, as opposed to the behaviour of the procedural version, `MultiplicationTable(~L)`, described above).

Note that this function is impractical in terms of CPU time and memory usage once $\dim(L)$ exceeds approximately 50. In such cases, the Lie algebra is more easily studied using the functions described in Section 106.4.3.

Example H106E13

We construct the generic Lie algebra generated by 3 extremal elements and construct a structure constant Lie algebra using the multiplication table.

```
> L := ExtremalLieAlgebra(Rationals(), 3);
> L:Maximal;
Lie algebra generated by 3 extremal elements, defined over Rational
Field
> MultiplicationTable(~L);
> L:Maximal;
Lie algebra generated by 3 extremal elements, originally defined over
Rational Field
  Now living over Polynomial ring of rank 4 over Rational Field
  Dimension: 8
  Picked 4 f-values:
    f(2, [1]) = f21
    f(3, [1]) = f31
```

```
    f(3, [2]) = f32
    f(1, [32]) = f132
> Dimension(L);
8
> MT := MultiplicationTable(L);
> MT[4][8];
(0   -1/2*f31*f32   0   -1/2*f132   0   0   0   1/2*f32)
> M := LieAlgebra<CoefficientRing(L), 8 | MT>;
> M;
Lie Algebra of dimension 8 with base ring Polynomial ring of rank 4
over Rational Field
> M.4*M.8;
(0   -1/2*f31*f32   0   -1/2*f132   0   0   0   1/2*f32)
> M.1*(M.1*M.2);
(f21   0   0   0   0   0   0   0)
```

106.4.3 Instances of Lie Algebras Generated by Extremal Elements

`Instance(L)`

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

The Lie algebra $L(f)$ for general f. The Lie algebra returned will in general be defined over a multivariate polynomial ring.

This function is identical to `MultiplicationTable`, except that it returns a Lie algebra rather than a multiplication table. Please refer to that function for information on the optional arguments `Rep` and `Check`. Note that this function also is impractical in terms of CPU time and memory usage once $\dim(L)$ exceeds approximately 50. In such cases, the Lie algebra is more easily studied by constructing particular instances of $L(f)$ individually, as described below.

`Instance(L, Q)`

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

Construct $L(f)$ where the i-th free parameter of X is set to Q[i]. Consult `L:Maximal` or `FreefValues` to obtain information about the free parameters. The coefficient ring of the Lie algebra M returned will be equal to `Universe(Q)`. As a second return value, an invertible map from M to the free Lie algebra of rank `Ngens(L)` is returned.

The optional argument `Rep` may be "Auto", "Dense" or "Sparse" (refer to the documentation at `MultiplicationTable` for more information). `Check` may be set to `true` or `false` and determines whether the Jacobi identity is checked on the Lie algebra returned.

Example H106E14

We construct the generic Lie algebra generated by 3 extremal elements and study one of its instances.

```
> L := ExtremalLieAlgebra(Rationals(), 3);
> MultiplicationTable(~L);
> L:Maximal;
Lie algebra generated by 3 extremal elements, originally defined over
Rational Field
  Now living over Polynomial ring of rank 4 over Rational Field
  Dimension: 8
  Picked 4 f-values:
    f(2, [1]) = f21
    f(3, [1]) = f31
    f(3, [2]) = f32
    f(1, [32]) = f132
 > M := Instance(L); M;
Lie Algebra of dimension 8 with base ring Polynomial ring of rank 4
over Rational Field
> M.1*(M.1*M.2);
(f21   0   0   0   0   0   0   0)
```

So in the most general case, $[x_1, [x_1, x_2]] = f_{x_2}(x_1)x_1$. Next, we consider an instance where we set $f_{x_2}(x_1) = 1/7$, $f_{x_3}(x_1) = 1/5$, $f_{x_3}(x_2) = 1/3$ and $f_{x_1}([x_3, x_2]) = 1$.

```
> N, phi := Instance(L, [Rationals()|1/7,1/5,1/3,1]);
> N;
Lie Algebra of dimension 8 with base ring Rational Field
> SemisimpleType(N);
A2
> N.1*(N.1*N.2);
(1/7   0   0   0   0   0   0   0)
> y := phi(N.2); z := phi(N.3);
> Parent(y):Minimal;
Free Lie algebra of rank 3 over Rational Field
> (y,(y,z));
-($.2, ($.3, $.2))
> (y,(y,z)) @@ phi;
(  0 1/3   0   0   0   0   0   0)
> (y,(y,z)) @@ phi @ phi;
1/3*$.2
```

106.4.4 Studying the Parameter Space

`FreefValues(L)`

The values $f_x(b)$ generating the parameter space X (see the introduction to this section 106.4 for details). This function returns two sequences: the first of the $f_x(b)$ as elements of `CoefficientRing(L)` and the second of the pairs (x, b) as two-tuples of integers.

`fValue(L, x, b)`

The value $f_x(b)$ as an element of `CoefficientRing(L)`.

`fValueProof(L, x, b)`

Print a proof of correctness for the value $f_x(b)$.

Example H106E15

We consider the generic Lie algebra generated by 4 extremal elements.

```
> L := ExtremalLieAlgebra(Rationals(), 4);
> vals, pairs := FreefValues(L);
> vals;
[
    f21,
    f31,
    f41,
    f32,
    f42,
    f43,
    f143,
    f243,
    f142,
    f132,
    f1432,
    f2431
]
> #vals;
12
> pairs;
[ <2, 1>, <3, 1>, <4, 1>, <3, 2>, <4, 2>, <4, 3>, <1, 5>, <2, 5>, <1,
6>, <1, 8>, <1, 11>, <2, 12> ]
```

This shows that $\dim(X) = 12$. We compute some values $f_x(b)$.

```
> fValue(L, 1, 5);
f143
> fValue(L, 4, 17);
-f41*f42
> fValueProof(L, 4, 17);
f(4, [241]) -> -f(4,[2])*f(4,[1]) {f(x,[y,[x,N]]) = -f(x,y)f(x,N) by
```

```
  assoc. of f and anti-comm. of L}
    f(4, [2]) = f42 {Free}
    f(4, [1]) = f41 {Free}
= -f41*f42
```

`DimensionsEstimate(L, g)`

`NumSamples`	RngIntElt	*Default :* ∞
`Check`	BoolElt	*Default :* `true`
`Rep`	MonStgElt	*Default :* "*Auto*"
Verbose	`AlgLieExtr`	*Maximum :* 10

Estimate the dimensions of the subvarieties of the parameter space X of L giving rise to irreducible Lie algebra modules of different dimensions.

This procedure repeatedly (exactly `NumSamples` times) invokes `Instance(L, g())` to produce a Lie algebra M. The composition series of M are computed, and the dimension e of its simple factor is stored. Then, for each of these e encountered, the dimension of the subvariety (inside the algebraic variety X) that contains Lie algebras whose top factor has dimension e is estimated using the dimension d of the full f-variety. (Here d is taken to be the number of free f-values computed; see `FreefValues`).

If the verbose flag `"AlgLieExtr"` is set 3 or more, then after each step the estimate is printed as a sequence of triples (e, n, s): n is the number of times dimension e was encountered, and s the estimate for the dimension of the subvariety.

Upon finishing (which will only happen if `NumSamples` is set to some finite number) that sequence of triples is returned. The second return value is a multiset containing the dimensions encountered in the search.

Note that this procedure assumes that X itself is an affine variety (which has been proved if `CommutatorGraph(L)` is a connected simply laced Dynkin diagram of finite or affine type) and that `g` produces uniformly random elements of X. If either of these two is not the case, the estimates produced are likely wrong. Moreover, `g` must produce sequences of elements of a finite field.

The optional argument `Rep` may be "Auto", "Dense" or "Sparse" (refer to the documentation at `MultiplicationTable` for more information). `Check` may be set to `true` or `false` and determines whether the Jacobi identity is checked on the Lie algebras constructed.

`InstancesForDimensions(L, g, D)`

`Check` BoolElt *Default :* `true`

For each $d \in D$ attempt to find an instance of L whose simple factor has dimension d, by repeatedly invoking `Instance(L, g())`. The result is returned in the form of an associative array A such that, for all $d \in D$, $A[d]$ is a triple (v, M, ϕ) where v is such that `Instance(L, v)` is M, and ϕ is an invertible map from M to the free Lie algebra.

See `DimensionsEstimate` for the required properties of `g`. The optional parameter `Check` may be set to `true` or `false` and determines whether the Jacobi identity is checked on the Lie algebras constructed.

Example H106E16

We consider the generic Lie algebra generated by 3 extremal elements.

```
> L := ExtremalLieAlgebra(Rationals(), 3);
> FreefValues(L);
[
    f21,
    f31,
    f32,
    f132
]
[ <2, 1>, <3, 1>, <3, 2>, <1, 4> ]
```

So $\dim(X) = 4$. We create a function `g` used to construct random instances of L over GF(5).

```
> g := func< | [ Random(GF(5)) : i in [1..4] ]>;
> repeat M := Instance(L, g()); until Rank(KillingForm(M)) eq 8;
> M;
Lie Algebra of dimension 8 with base ring GF(5)
> SemisimpleType(M);
A2
```

So in this case `g()` yielded a Lie algebra of type A_2. We use `g` to obtain information about X, using 500 random instances.

```
> DimensionsEstimate(L, g : NumSamples := 500);
[ <3, 121, "3.12">, <8, 379, "3.83"> ]
{* 3^^121, 8^^379 *}
```

This shows that 379 instances were found where M was simple of dimension 8, and 121 cases where M had a simple factor of dimension 3. Using this result one might conjecture that there is a codimension 1 subspace of X with Lie algebras whose simple factor has dimension 3.

```
> A := InstancesForDimensions(L, g, {3,8} : Check := false);
> A[3];
<[ 2, 1, 4, 4 ], Lie Algebra of dimension 8 with base ring GF(5),
Mapping from: Lie Algebra of dimension 8 with base ring GF(5) to Free
Lie algebra of rank 3 over GF(5) given by a rule>
```

```
> M := A[3][2]; MM := M/SolvableRadical(M); MM;
Lie Algebra of dimension 3 with base ring GF(5)
> SemisimpleType(MM);
A1
> M := A[8][2]; IsSimple(M);
true
> SemisimpleType(M);
A2
```

106.5 Families of Lie Algebras

The radical of a Lie algebra is the maximal soluble ideal. A Lie algebra is called *reductive* if its radical is equal to its centre, and *semisimple* if its radical is trivial. A Lie algebra is *almost reductive (resp. simple, semisimple)* if the corresponding group of Lie type is reductive (resp. simple, semisimple). Note that these concepts are equivalent if the field has characteristic zero.

The commands in this section construct almost reductive Lie algebras over an arbitrary field. Such Lie algebras have a corresponding root datum. The matrix versions of these commands give the standard matrix representation, which is the smallest degree representation (with a few exceptions for small characteristic fields).

106.5.1 Almost Reductive Lie Algebras

The intrinsics `LieAlgebra` and `MatrixLieAlgebra` described below take as first argument an object which describes the type of the reductive Lie algebra to be constructed. Specifically, it may be one of the five following types:

(a) A string describing the Cartan type;

(b) A root datum (see Chapter 103);

(c) A crystallographic root system (see Chapter 102);

(d) A Dynkin digraph (see Section 101.5);

(e) A crystallographic Cartan matrix C (see Section 101.4).

In the cases (a), (d), and (e) these intrinsics take an optional argument `Isogeny`. See Section 109.2 for the possible values of this flag.

`LieAlgebra(T, k)`

`Isogeny` . *Default :* "*Ad*"

Construct the reductive Lie algebra of type T over the ring k.

`MatrixLieAlgebra(T, k)`

`Isogeny` . *Default :* "*Ad*"

Construct the reductive matrix Lie algebra of type T over the ring k.

Example H106E17

We construct some (semi)simple Lie algebras.

```
> LieAlgebra("D7", RationalField());
Lie Algebra of dimension 91 with base ring Rational Field
> LieAlgebra("G2", GF(5));
Lie Algebra of dimension 14 with base ring GF(5)
> L := LieAlgebra( "G2 B3", Rationals() );
> L;
Lie Algebra of dimension 35 with base ring Rational Field
> DirectSumDecomposition(L);
[
    Lie Algebra of dimension 14 with base ring Rational Field,
    Lie Algebra of dimension 21 with base ring Rational Field
]
> LieAlgebra( "E8", GF(2) );
Lie Algebra of dimension 248 with base ring GF(2)
```

Example H106E18

This example demonstrates the use of the `Isogeny` option. Over a field of characteristic zero, this option only effects the basis used. In characteristic p, it sometimes effects the isomorphism type of the algebra. For type A_n with $p|(n+1)$, the default Isogeny is "Ad" (adjoint), which gives an algebra with nontrivial derived subalgebra but no centre:

```
> L := LieAlgebra("A4", GF(5));
> Dimension(L);
24
> Dimension(L*L);
23
> Dimension(Centre(L));
0
```

If you take Isogeny to be "SC" (simply connected), you get a perfect algebra with a nontrivial centre.

```
> L := LieAlgebra("A4", GF(5) : Isogeny:="SC");
> Dimension(L);
24
> Dimension(L*L);
24
> Dimension(Centre(L));
1
```

If $p^2|(n+1)$ there is an intermediate isogeny type which has both a centre and a nontrivial derived algebra:

```
> L := LieAlgebra("A24", GF(5) : Isogeny:=5);
> Dimension(L);
624
```

```
> Dimension(L*L);
623
> Dimension(Centre(L));
1
```

Similar results can be obtained by constructing the Lie algebra from a root datum. This kind of phenomenon happens whenever the characteristic divides the order of the fundamental group of your root datum. See [Hog82] for more details.

```
> R := RootDatum("E6");
> #FundamentalGroup(R);
3
> L := LieAlgebra(R,GF(3));
> L;
Lie Algebra of dimension 78 with base ring GF(3)
> L*L;
Lie Algebra of dimension 77 with base ring GF(3)
```

`LieAlgebra(N, k, p)`

`LieAlgebra(R, k, p)`

The twisted (almost) semisimple Lie algebra over the finite field k with Cartan type N given as a string or root datum R, with twist given by the permutation p. The twist should either be a permutation of the indices of the simple roots, or of the indices of all roots.

`TwistedLieAlgebra(R, k)`

Given a twisted root datum R and a finite field k, construct the twisted Lie algebra $L = R(k)$.

This variant has 5 return values. First, the twisted Lie algebra L. Second, a homomorphism ϕ from L into the split Lie algebra L' (over a suitable field extension of k); Third, L'; Fourth, a split toral subalgebra H of L, and, fifth, a split toral subalgebra H' of L', such that $\phi(H) \subseteq H'$.

See also `TwistedBasis`.

Example H106E19

We construct two twisted Lie algebras.

```
> DynkinDiagram("E6");
E6    1 - 3 - 4 - 5 - 6
              |
              2
> LieAlgebra( "E6", GF(5), Sym(6)!(1,6)(3,5) );
Lie Algebra of dimension 78 with base ring GF(5)
> Rt := TwistedRootDatum(RootDatum("D4") : Twist := 3);
> k := GF(7);
```

```
> L, phi, Lp, H, Hp := TwistedLieAlgebra(Rt, k);
> L;
Lie Algebra of dimension 28 with base ring GF(7)
> Lp;
Lie Algebra of dimension 28 with base ring GF(7^3)
> phi(L.3);
(0 0 ksi^49 ksi^7 ksi 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0)
> IsSplitToralSubalgebra(L, H);
true
> IsSplitToralSubalgebra(Lp, Hp);
true
> forall{b : b in Basis(H) | phi(b) in Hp};
true
```

106.5.2 Cartan-Type Lie Algebras

Simple Lie algebras over fields of characteristic 0 have been classified and are precisely the twisted forms of Lie algebras of types $A_l, B_l, C_l, D_l, E_6, E_7, E_8, F_4$ and G_2 (see previous Subsection). Over fields of finite characteristic p, the analogues of these algebras are called *classical-type* (including the exceptional algebras). Over such fields there are other simple Lie algebras, the first of them found by Witt sometimes before 1937. For $p \geq 7$, the only non-classical simple Lie algebras are the Lie algebras of *Cartan-type*, which we discuss in this section. For $p = 5$, one further class of simple Lie algebras occurs: *Melikian* algebras, which are discussed in the next section. In characteristic 2 and 3, the classification of simple Lie algebras is not yet complete.

Cartan-type Lie algebras are non-classical Lie algebras which arise from infinite dimensional algebras of differential operators over $\mathbf{C}$:

- (generalised) Witt algebras,
- special and conformal special Lie algebras,
- Hamiltonian and conformal Hamiltonian Lie algebras,
- and contact Lie algebras.

The notation and the description of these Lie algebras closely follow Strade and Farnsteiner [Str04] and [SF88]. Where the notation of the two books differs, we follow [Str04].

Let F be a finite field of characteristic $p > 0$ and m a positive integer. We refer for the definition of $O(m)$ and $x^{(a)}$ to [Str04, 2.1]. The basis of $O(m)$ is $\{x^{(a)} | 0 \leq a, a \in \mathbf{N}^m\}$.

Let n be a sequence of positive integers of length m and set $N := \sum_{i=1}^{m} n_i$. Define

$$O(m, n) := \langle x^{(a)} | 0 \leq a_i < p^{n_i} \rangle$$

For $i = 1, \ldots, m$ denote by ∂_i the derivation of $O(m)$ defined by

$$\partial_i(x_j^{(r)}) = \delta_{i,j} x_j^{(r-1)}.$$

Now define

$$W(m,n) := \sum_{i=1}^{m} O(m)\partial_i.$$

The algebra $W(m,n)$ is the *Witt* algebra and has dimension mp^N over F. In particular, $W(1,[1])$ is the standard p-dimensional Witt algebra.

The Witt algebra $W(m,n)$ is simple unless $p=2$ and $m=1$ ([SF88, 4.2.4(1)]) and is restrictable if and only if $n=[1,\ldots,1]$ ([SF88, 4.2.4(2)]).

Further define

$$\Omega^0(m,n) := O(m,n),$$

$$\Omega^1(m,n) := \mathrm{Hom}_{O(m,n)}(W(m,n), O(m,n)),$$

$$\Omega^r(m,n) := \bigwedge^r \Omega^1(m,n),$$

$$\Omega(m,n) := \bigoplus \Omega^r(m,n).$$

Let $m \geq 2$ and $\omega_S = dx_1 \wedge \ldots \wedge dx_m$. Define the following subalgebras of $W(m,n)$:

$$S(m,n) := \{D \in W(m,n) | D(\omega_S) = 0\},$$

$$CS(m,n) := \{D \in W(m,n) | D(\omega_S) \in F\omega_S\}.$$

The algebra $S(m,n)$ is the *special* and $CS(m,n)$ is the *conformal special* Lie algebra. The dimension of $S(m,n)$ over F is $(m-1)p^N + 1$ and the dimension of $CS(m,n)$ is $\dim S(m,n) + 1$.

Suppose $m \geq 3$. Then the algebra $S(m,n)^{(1)}$ is simple ([SF88, 4.3.5(1)]) and is restrictable if and only if $n=[1,\ldots,1]$ ([SF88, 4.3.5(2)]).

Let $p>2$, $m=2r\geq 2$ and let $\omega_H = \sum_{i=1}^{r} dx_i \wedge dx_{i+r}$. Define the following subalgebras of $W(m,n)$:

$$H(m,n) := \{D \in W(m,n) | D(\omega_H) = 0\},$$

$$CH(m,n) := \{D \in W(m,n) | D(\omega_H) \in F\omega_H\}.$$

The algebra $H(m,n)$ is the *Hamiltonian* and $CH(m,n)$ is the *conformal Hamiltonian* Lie algebra. The dimension of $H(m,n)$ over F is $p^N - 1$ and the dimension of $CH(m,n)$ is $\dim H(m,n) + 1$.

The algebra $H(m,n)^{(2)}$ is simple ([SF88, 4.4.5(1)]) and is restrictable if and only if $n=[1,\ldots,1]$ ([SF88, 4.4.5(2)]). And, if $m>2$, then $H(m,n)^{(2)} = H(m,n)^{(1)}$.

Let $p>2$, $m=2r+1\geq 3$ and let $\omega_K = dx_m + \sum_{i=1}^{r}(x_i dx_{i+r} - x_{i+r}dx_i)$. Define the following subalgebra of $W(m,n)$:

$$K(m,n) := \{D \in W(m,n) | D(\omega_K) \in O(m,n)\omega_K\},$$

The algebra $K(m,n)$ is the *contact* Lie algebra. The dimension of $K(m,n)$ over F is p^N.

The algebra $K(m,n)^{(1)}$ is simple ([SF88, 4.5.5(1)]) and is restrictable if and only if $n=[1,\ldots,1]$ ([SF88, 4.5.6]). If $m+3 \not\equiv 0 \bmod p$, then $K(m,n)^{(1)} = K(m,n)$.

`WittLieAlgebra(F, m, n)`

Check BoolElt *Default :* false

The Witt algebra $W(m, n)$ is constructed over the finite field F, where m must be a positive integer and n a sequence of positive integers of length m. If the optional argument Check is true, the algebra is checked to be Lie upon construction.

An invertible map from the polynomial ring P over F of degree $2m$ to $W(m, n)$ is returned as second value, to assist in identifying the elements of $W(m, n)$. For $1 \leq i \leq m$ the i-th generator of P maps to x_i in $W(m, n)$, and for $m+1 \leq i \leq 2m$ the i-th generator of P maps to δ_{i-m} in $W(m, n)$.

Example H106E20

We compute the Witt algebra $W(2, [2, 1])$ over $GF(9)$ and verify the multiplication of $x_1^{(1)}\delta_1$ and $x_1^{(2)}x_2^{(1)}\delta_2$.

```
> W, phi := WittLieAlgebra(GF(9), 2, [2,1]);
> W;
Lie Algebra of dimension 54 with base ring GF(3^2)
> IsSimple(W);
true
> P<x1, x2, d1, d2> := Domain(phi);
> phi(x1*d1);
(0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
    0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0)
> (phi(x1*d1)*phi(x1^2*x2*d2)) @@ phi;
2*x1^2*x2*d2
```

and the standard Witt algebra $W(1, [1])$ over $GF(2)$:

```
> W := WittLieAlgebra(GF(2), 1, [1]);
> W;
Lie Algebra of dimension 2 with base ring GF(2)
> IsSimple(W);
false
> IsRestrictedLieAlgebra(W);
true [ (0 0), (0 1) ]
```

`SpecialLieAlgebra(F, m, n)`

`ConformalSpecialLieAlgebra(F, m, n)`

Check BoolElt *Default :* false

The (conformal) special Lie algebra $(C)S(m, n)$ is constructed over the finite field F, where $m \geq 2$ must be an integer and n a sequence of positive integers of length m. If the optional argument Check is true, Magma checks that the algebra constructed is a Lie algebra.

The intrinsic `SpecialLieAlgebra` returns the Witt algebra $W(m, n)$ in which it is embedded as the second return value. In addition, similarly to `WittLieAlgebra`, a map from the polynomial ring P of degree $2m$ over F to $S(m, n)$ is returned as the third return value, and a map from P to $W(m, n)$ as the fourth return value.

Similarly, `ConformalSpecialLieAlgebra` returns the special Lie algebra $S(m, n)$ which it contains and the Witt Lie algebra $W(m, n)$ in which it is embedded in as second and third return values. Maps from P to $CS(m, n)$, $S(m, n)$, and $W(m, n)$ are returned as fourth, fifth, and sixth return values, respectively.

Example H106E21

We compute both $S(3, [1, 2, 1])$ and $CS(3, [1, 2, 1])$ over $GF(9)$:

```
> CS,S,W := ConformalSpecialLieAlgebra( GF(9), 3, [1,2,1] );
> CS;S;W;
Lie Algebra of dimension 164 with base ring GF(3^2)
Lie Algebra of dimension 163 with base ring GF(3^2)
Lie Algebra of dimension 243 with base ring GF(3^2)
> IsSimple(S);
false
> IsSimple(S*S);
true
> IsRestrictedLieAlgebra(S*S);
false []
```

`HamiltonianLieAlgebra(F, m, n)`

`ConformalHamiltonianLieAlgebra(F, m, n)`

`Check` BoolElt *Default :* `false`

The (conformal) Hamiltonian Lie algebra $(C)H(m, n)$ is constructed over the finite field F of characteristic at least 3, where $m \geq 2$ must be even and n a sequence of positive integers of length m. If the optional argument `Check` is `true`, the algebra is checked to be Lie upon construction.

The intrinsic `HamiltonianlLieAlgebra` returns the Witt Lie algebra $W(m, n)$ in which it is embedded as the second return value. Additionally, similarly to `WittLieAlgebra`, a map from the polynomial ring P of degree $2m$ over F to $H(m, n)$ is returned as the third return value, and a map from P to $W(m, n)$ as the fourth return value.

Similarly, `ConformalHamiltonianLieAlgebra` returns the Hamiltonian Lie algebra $H(m, n)$ it contains and the Witt Lie algebra $W(m, n)$ in which it is embedded as the second and third return values. Maps from P to $CH(m, n)$, $H(m, n)$, and $W(m, n)$ are returned as the fourth, fifth, and sixth return values, respectively.

Example H106E22

We compute both $H(2,[2,2])$ and $CH(2,[2,2])$ over $GF(9)$:

```
> CH,H,W := ConformalHamiltonianLieAlgebra( GF(9), 2, [2,2] );
> CH;H;W;
Lie Algebra of dimension 81 with base ring GF(3^2)
Lie Algebra of dimension 80 with base ring GF(3^2)
Lie Algebra of dimension 162 with base ring GF(3^2)
> IsSimple(H);
false
> IsSimple(H*H);
true
> IsSimple(H*H*H);
true
> IsRestrictedLieAlgebra(H*H*H);
false []
```

`ContactLieAlgebra(F, m, n)`

`Check` BoolElt *Default :* `false`

The contact Lie algebra $K(m,n)$ is constructed over the finite field F of characteristic at least 3, where $m \geq 3$ must be odd and n a sequence of positive integers of length m. If the optional argument `Check` is `true`, the algebra is checked to be Lie upon construction.

The intrinsic `ContactLieAlgebra` returns the Witt Lie algebra $W(m,n)$ in which it is embedded as the second return value. Additionally, similarly to `WittLieAlgebra`, a map from the polynomial ring P of degree $2m$ over F to $K(m,n)$ is returned as the third return value, and a map from P to $W(m,n)$ as the fourth return value.

Example H106E23

We compute the contact Lie algebra $K(3,[1,1,1])$ over $GF(5)$:

```
> K,W := ContactLieAlgebra( GF(5), 3, [1,1,1] );
> K;W;
Lie Algebra of dimension 125 with base ring GF(5)
Lie Algebra of dimension 375 with base ring GF(5)
> K*K eq K;
true
> IsSimple(K);
true
```

106.5.3 Melikian Lie Algebras

The Melikian Lie Algebras are a class of simple Lie algebras over finite fields of characteristic 5, parameterized by two positive integers n_1, n_2. We follow the explicit construction by Strade [Str04, Section 4.3].

Let F be a field of characteristic $p = 5$ and recall the definition of $O(m, n)$ and $W(m, n)$ from Section 106.5.2. Define $W = W(2, [n_1, n_2])$, $O = O(2, [n_1, n_2])$, and take W' to be a copy of W. We equip the vector space $W \oplus O \oplus W'$ with a bilinear product $[\cdot, \cdot]$ that is defined by the following equations, where $D, E \in W$ and $f, f_1, f_2, g, g_1, g_2 \in O$.

- On $W \times W$, the usual multiplication in W.
- On $W \times O$: $[D, f] = D(f) - 2\mathrm{div}(D)f$.
- On $W \times W'$: $[D, E'] = ([D, E])' + 2\mathrm{div}(D)E'$.
- On $O \times O$: $[f, g] = 2(g\delta_2(f) - f\delta_2(g))\delta_1' + 2(f\delta_1(g) - g\delta_1(f))\delta_2'$.
- On $O \times W'$: $[f, E'] = fE$.
- On $W' \times W'$: $[f_1\delta_1' + f_2\delta_2', g_1\delta_1' + g_2\delta_2'] = f_1g_2 - f_2g_1$.

Here div is the linear map defined by $\mathrm{div}(f\delta_i) = \delta_i f$. It follows that $M(n_1, n_2)$, of dimension $5^{n_1+n_2+1}$, is a simple Lie algebra [Str04, Lemma 4.3.1, Theorem 4.3.3].

`MelikianLieAlgebra(F, n1, n2)`

`Check` BoolElt *Default :* `false`

The Melikian Lie algebra $M = M(n_1, n_2)$ over F. An invertible map from the polynomial ring P of degree 6 over F to M is returned as second value, to assist in identifying the elements of M. Here the six generators of P represent x_1, x_2, δ_1, δ_2, δ_1', δ_2', respectively.

Example H106E24

We construct $M(2, 1)$ over $\mathbf{F}_5$ and inspect some of its properties.

```
> M, phi := MelikianLieAlgebra(GF(5), 2, 1);
> M;
Lie Algebra of dimension 625 with base ring GF(5)
> IsSimple(M);
true
```

Next, we construct subspaces (not subalgebras) W, O, W' of M.

```
> P<x1, x2, d1, d2, dp1, dp2> := Domain(phi);
> V := VectorSpace(GF(5), Dimension(M));
> W := sub<V | [ V | phi(x1^i*x2^j*d) : i in [0..24], j in [0..4],
>                                                    d in [d1,d2] ]>;
> O := sub<V | [ V | phi(x1^i*x2^j) : i in [0..24], j in [0..4] ]>;
> Wp := sub<V | [ V | phi(x1^i*x2^j*d) : i in [0..24], j in [0..4],
>                                                     d in [dp1,dp2] ]>;
> Dimension(W), Dimension(O), Dimension(Wp);
```

```
250 125 250
> Dimension(W meet O), Dimension(W meet Wp), Dimension(O meet Wp);
0 0 0
```

Finally, we verify that these subspaces multiply as required by the definition.

```
> m := func< A, B | sub<V | [ V | M!a*M!b : a in Basis(A), b in Basis(B) ]> >;
> WxWp := m(W, Wp); [ WxWp subset VV : VV in [W, O, Wp] ];
[ false, false, true ]
```

So indeed $[W, W'] \subseteq W'$.

```
> VV := [W, O, Wp]; VVnm := ["W", "O", "W'" ];
> mm := function(A, B)
>   AB := m(A, B);
>   for i in [1..#VV] do
>     if AB eq VV[i] then return VVnm[i]; end if;
>   end for;
>   return "??";
> end function;
> mm(W, Wp);
W'
> for i,j in [1..#VV] do
>   printf "[ %2o, %2o ] = %2o%o", VVnm[i], VVnm[j], mm(VV[i], VV[j]),
>     (j eq 3) select "\n" else ", ";
> end for;
[  W,  W ] =  W, [  W,  O ] =  O, [  W, W' ] = W'
[  O,  W ] =  O, [  O,  O ] = W', [  O, W' ] =  W
[ W',  W ] = W', [ W',  O ] =  W, [ W', W' ] =  O
```

106.6 Construction of Elements

`Zero(L)`

`L ! 0`

The zero element of the Lie algebra L.

`Random(L)`

Given a Lie algebra L defined over a finite ring, a random element is returned.

106.6.1 Construction of Elements of Structure Constant Algebras

`elt< L | r1,r2,...,rn >`

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

`L ! Q`

Given a Lie algebra L of dimension n and a sequence $Q = [r_1, r_2, \ldots, r_n]$ of elements of the base ring R of L, the element $r_1 * e_1 + r_2 * e_2 + \ldots + r_n * e_n$ of L is constructed.

`BasisProduct(L, i, j)`

Returns the product of the i-th and j-th basis element of the Lie algebra L.

`BasisProducts(L)`

`Rep` MonStgElt *Default : "Dense"*

Returns the products of all basis elements of the Lie algebra L.

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

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

106.6.2 Construction of Matrix Elements

Matrix Lie elements can be constructed using the functions below. For more information on constructing matrices see Section 88.2.2.

`elt< R | L >`

`R ! L`

Create the element of the matrix Lie algebra R of degree n whose entries are the n^2 elements of the sequence L.

`DiagonalMatrix(L, Q)`

Diagonal matrix in the matrix Lie algebra L, given by the sequence Q of ring elements.

`ScalarMatrix(L, r)`

Scalar matrix in the matrix Lie algebra L, defined by the ring element r.

106.7 Construction of Subalgebras, Ideals and Quotients

If the coefficient ring R of a Lie algebra L is a Euclidean domain, then submodules and ideals can be constructed in MAGMA; if R is a field then quotients can be constructed in MAGMA. Note that left, right, and two-sided ideals are identical in a Lie algebra.

`sub< L | A >`

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

(a) An element of L;

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

(c) A subalgebra or ideal of L;

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

As well as the subalgebra S itself, the constructor returns the inclusion homomorphism $f : S \to L$.

`ideal< L | A >`

Creates the ideal I of the Lie algebra L generated by the elements defined by A, where A is a list of one or more items of the following types:

(a) An element of L;

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

(c) A subalgebra or ideal of L;

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

As well as the ideal I itself, the constructor returns the inclusion homomorphism $f : I \to L$.

`quo< L | A >`

Forms the quotient algebra L/I, where I is the two-sided ideal of L generated by the elements defined by A, where A is a list of one or more items of the following types:

(a) An element of L;

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

(c) A subalgebra or ideal of L;

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

As well as the quotient L/I itself, the constructor returns the natural homomorphism $f : L \to L/I$.

`L / S`

The quotient of the Lie algebra L by the ideal closure of the subalgebra S.

Example H106E25

We construct the quotient of the matrix Lie algebra of 2×2 matrices, by the ideal spanned by the identity matrix.

```
> L := MatrixLieAlgebra( Rationals(), 2 );
> Dimension(L);
4
> I := ideal< L | L!Matrix([[1,0],[0,1]]) >;
> Dimension(I);
1
> K := L/I;
> Dimension(K);
3
> SemisimpleType( K );
A1
```

`QuotientWithPullback(L, I)`

Given a Lie algebra L and an ideal I of L, the quotient L/I is constructed. As second return value, the natural homomorphism $f : L \to L/I$ is returned.

As third return value, a function g is returned. This g takes an element $y \in I$ and returns an $x \in L$ and a vector space V such that $f(x + v) = y$ for all $v \in V$. As fourth return value, a function h is returned. This h takes an element $y \in I$ and returns the subalgebra of L generated by x and V, with x and V as above.

Example H106E26

We consider an ideal of the Lie algebra of type G_2 over the field with 3 elements.

```
> R := RootDatum("G2");
> L := LieAlgebra(R, GF(3));
> pos,neg,cart := StandardBasis(L);
> shrt := [ i : i in [1..NumPosRoots(R)] | IsShortRoot(R, i) ];
> shrt;
[ 1, 3, 4 ]
> I := ideal<L | pos[shrt]>;
> _, str1 := ReductiveType(I); str1;
The 7-dim simple constituent of a Lie algebra of type A2
```

So apparently I is isomorphic to the 7-dimensional simple constituent of a Lie algebra of type A_2. We will now use `QuotientWithPullback` to construct L/I.

```
> LI, proj, pb, pbsub := QuotientWithPullback(L, I);
> _, str2 := ReductiveType(LI); str2;
The 7-dim simple constituent of a Lie algebra of type A2
```

So apparently $I \simeq L/I$! Finally, we will demonstrate the use of the additional return values. First, we verify that an element of I maps to 0 in L/I:

```
> proj(pos[1]);
```

```
(0 0 0 0 0 0 0)
```

And then we consider the preimage in L of a randomly chosen element of L/I.

```
> y := LI![0,1,1,1,1,0,1];
> y;
(0 1 1 1 1 0 1)
> x, V := pb(y);
> x;
(0 1 0 0 1 0 0 1 0 1 0 0 0 1)
> #V;
2187
> assert #V eq #I;
> {* proj(x + v) eq y : v in V *};
{* true^^2187 *}
```

So indeed $x + v$ is a preimage of y for all $v \in V$.

```
> M := pbsub(y);
> M, M meet I;
Lie Algebra of dimension 8 with base ring GF(3)
Lie Algebra of dimension 7 with base ring GF(3)
> _,str3 := ReductiveType(M);
> str3;
Twisted Lie algebra of type 2A2 [Ad]
```

106.8 Operations on Lie Algebras

`L eq K`

Returns `true` if, and only if, the Lie algebras L and K are equal.

`L ne K`

Returns `true` if, and only if, the Lie algebras L and K are not equal.

`L subset K`

Returns `true` if, and only if, the Lie algebra L is contained in the Lie algebra K.

`L notsubset K`

Returns `true` if, and only if, the Lie algebra L is not contained in the Lie algebra K.

`L meet M`

The intersection of the Lie algebras L and M is returned. Note that L and M have a common superalgebra.

`L * M`

The Lie algebra product $[L, M]$ of the algebras L and M is returned. Note that L and M must have a common superalgebra.

`L ^ n`

The (left-normed) n-th power of the (structure constant) Lie algebra L, i.e., $((\ldots(L*L)*\ldots)*L)$ is constructed.

`Morphism(L, M)`

The map giving the morphism from the (structure constant) Lie algebra L to M is constructed. Either L is a subalgebra of M, in which case the embedding of L into M is returned, or M is a quotient algebra of L, in which case the natural epimorphism from L onto M is returned.

`IsIsomorphic(L, M)`

`HL`	AlgLie	*Default :* `false`
`HM`	AlgLie	*Default :* `false`

Returns `true` if the Lie algebras L and M are isomorphic. It is currently implemented for trivial cases (such as when the dimensions differ), reductive Lie algebras, solvable Lie algebras up to dimension 4, nilpotent Lie algebras up to dimension 6 (some special cases excluded). The solvable and nilpotent cases are handled using the databases for such algebras described in Section 106.18).

In the case of reductive Lie algebras, split maximal toral subalgebras for L and M may be provided in the optional arguments HL and HM, respectively. If these are not provided an attempt is made to compute them, a process which may fail, particularly in characteristic 0.

This intrinsic has two return values: the first a boolean describing whether L and M are isomorphic. If so, the second is an isomorphism from L to M, otherwise the second is a string describing the reason for non-isomorphism.

An error is thrown if isomorphism cannot be determined.

`IsKnownIsomorphic(L, M)`

`HL`	AlgLie	*Default :* `false`
`HM`	AlgLie	*Default :* `false`

Returns `true` if Magma can determine isomorphism between Lie algebras L and M. If so, the second return value is whether L and M are isomorphic, and the third is an isomorphism or a string (describing the reason for non-isomorphism). Refer to IsIsomorphic for more details on applicability and the meanings of the return values.

`IsIsomorphism(m)`

Returns `true` if the mapping m between two Lie algebras is an isomorphism of Lie algebras.

Example H106E27

We demonstrate that B_2 and C_2 are isomorphic over $\mathbf{Q}$.

```
> k := Rationals();
> L := LieAlgebra("B2", k); M := LieAlgebra("C2", k);
> b, c := IsIsomorphic(L, M);
> b;
true
> IsIsomorphism(c);
true
> c(L.1);
(0 0 1 0 0 0 0 0 0 0)
```

We demonstrate that B_3 and C_3 are non-isomorphic over $\mathbf{Q}$.

```
> L := LieAlgebra("B3", k); M := LieAlgebra("C3", k);
> b, c := IsIsomorphic(L, M);
> b;
false
> c;
21-dim component of L1 of type R1: Adjoint root datum of dimension 3 of type B3
didn't match R2:
Adjoint root datum of dimension 3 of type C3
```

We demonstrate that two distinct isogenies of B_2 are isomorphic over $\mathbf{Q}$.

```
> L := LieAlgebra("B2", k : Isogeny := "Ad");
> M := LieAlgebra("B2", k : Isogeny := "SC");
> b, c := IsIsomorphic(L, M);
> b;
true
```

For larger nilpotent algebras MAGMA cannot decide on the isomorphism question.

```
> L := LieAlgebra("B4", k);
> pL, _, _ := StandardBasis(L);
> subL := sub<L | pL>;
> subL;
Lie Algebra of dimension 16 with base ring Rational Field
> M := LieAlgebra("C4", k);
> pM, _, _ := StandardBasis(M);
> subM := sub<M | pM>;
> subL;
Lie Algebra of dimension 16 with base ring Rational Field
> IsNilpotent(subL), IsNilpotent(subM);
true true
> a,b,c := IsKnownIsomorphic(subL, subM);
> a;
false
```

Example H106E28

We demonstrate that in characteristic 3 the Lie algebras of type G_2 and A_2 have isomorphic nontrivial ideals.

```
> k := GF(3);
> CSL := CompositionSeries(LieAlgebra("G2", k));
> CSL;
[
    Lie Algebra of dimension 7 with base ring GF(3),
    Lie Algebra of dimension 14 with base ring GF(3)
]
> L := CSL[1];
> CSM := CompositionSeries(LieAlgebra("A2", k));
> CSM;
[
    Lie Algebra of dimension 7 with base ring GF(3),
    Lie Algebra of dimension 8 with base ring GF(3)
]
> M := CSM[1];
> a,b,c := IsKnownIsomorphic(L, M);
> a;
true
> b, c;
true Mapping from: AlgLie: L to AlgLie: M given by a rule
> IsIsomorphism(c);
true
```

106.8.1 Basic Invariants

`CoefficientRing(L)`

`BaseRing(L)`

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

`Dimension(L)`

The dimension of the Lie algebra L.

`#L`

The cardinality of the Lie algebra L, if the coefficient ring is finite.

`Moduli(L)`

This returns a sequence of integers, of length equal to the dimension of L. If the i-th element of this sequence is a_i then a_i is the minimal non-negative integer such that $a_i e_i = 0$. So if L is defined over a field, then the sequence consists of zeros.

Example H106E29

```
> T:= [ <1,2,2,2>, <2,1,2,2> ];
> t:= [0,4];
> L:= LieAlgebra< t | T : Rep:= "Dense" >;
> Moduli(L);
[ 0, 4 ]
```

106.8.2 Changing Base Rings

`ChangeRing(L, S)`

Given a Lie algebra L with base ring R, together with a ring S, this function constructs the Lie algebra M with base ring S obtained by coercing the coefficients of elements of L into S. The homomorphism from L to M is produced as second return value.

`ChangeRing(L, S, f)`

Given a Lie algebra L with base ring R, together with a ring S and a map $f : R \to S$, this function constructs the Lie algebra M with base ring S obtained by mapping the coefficients of elements of L into S via f. The homomorphism from L to M is produced as the second return value.

106.8.3 Bases

`BasisElement(A, i)`

`A . i`

The i-th basis element of the algebra L.

`Basis(A)`

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

`IsIndependent(Q)`

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

`ExtendBasis(S, L)`

`ExtendBasis(Q, L)`

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

106.8.4 Operations for Semisimple and Reductive Lie Algebras

`SemisimpleType(L)`

`CartanName(L)`

Let L be a Lie algebra. If L has a nondegenerate Killing form, then (over some algebraic extension of the ground field) L is the direct sum of absolutely simple Lie algebras. These Lie algebras have been classified and the classes are named A_n, B_n, C_n, D_n, E_6, E_7, E_8, F_4 and G_2. This function returns a single string containing the types of the direct summands of L.

For a description of the algorithm used in the general case we refer to [dG00], §5.17.1. For Lie algebras over fields of characteristic 2 and 3 the algorithm used is described in [Roo10], Chapter 5.

Example H106E30

We compute the semisimple type of the Levi subalgebra of a subalgebra of the simple Lie algebra of type D_7.

```
> L := LieAlgebra("D7", RationalField());
> L;
Lie Algebra of dimension 91 with base ring Rational Field
> K := Centralizer(L, sub<L | [L.1,L.2,L.3,L.4]>);
> K;
Lie Algebra of dimension 41 with base ring Rational Field
> _,S := HasLeviSubalgebra(K);
> S;
Lie Algebra of dimension 6 with base ring Rational Field
> SemisimpleType(S);
A1 A1
```

`ReductiveType(L)`

`ReductiveType(L, H)`

`AssumeAlmostSimple` BoolElt *Default :* `false`

Let L be a Lie algebra of a reductive algebraic group, and H a split maximal toral subalgebra of L. This function identifies the isomorphism type of L.

This function has four return values. The first is the appropriate root datum and the second return value a textual description of L. The third return value is a sequence Q, containing a decomposition of L into direct summands. Finally, the fourth return value is a sequence P of records, such that $P[i]$ contains additional information (often a proof of correctness) of the identification of $Q[i]$.

If a split maximal toral subalgebra H is not given, an attempt is made to compute one by calling `SplitMaximalToralSubalgebra` if the characteristic of the base field k is at least 5, or `SplitToralSubalgebra` if $\mathrm{char}(k)$ is 2 or 3. Note that, if k is

infinite, such a subalgebra cannot in general be computed so the second parameter H must be supplied for this function to work.

If the optional parameter `AssumeAlmostSimple` is set to true, the (possibly time consuming) step of computing a direct sum decomposition of L is skipped.

Moreover, note that if L is the Lie algebra of a simple algebraic group but itself non-simple (such as for example A_n of intermediate type in characteristic $n+1$), the third return value Q may not be the direct sum decomposition of L but simply $[L]$.

Example H106E31

We consider a particular Lie algebra of type A_3 over $k = \mathrm{GF}(2)$.

```
> RA3 := RootDatum("A3" : Isogeny := 2);
> L := LieAlgebra(RA3, GF(2));
> D := DirectSumDecomposition(L);
> D;
[
    Lie Algebra of dimension 14 with base ring GF(2),
    Lie Algebra of dimension 1 with base ring GF(2)
]
> R, str, Q, _ := ReductiveType(L);
> R;
RA3: Root datum of dimension 3 of type A3
> str;
Lie algebra of type A3[ 2]
> Q;
[
    Lie Algebra of dimension 15 with base ring GF(2)
]
```

Note that this is an example where Q is not the direct sum decomposition of L. Instead, L in its whole is recognised as the Lie algebra of a simple algebraic group. In the remainder of the example, we investigate the 14-dimensional ideal of L.

```
> M := D[1]; M;
Lie Algebra of dimension 14 with base ring GF(2)
> R, _, _, P := ReductiveType(M);
> R;
R: Adjoint root datum of dimension 2 of type G2
```

So this computation claims that $L \simeq M \oplus k$, where M is of type G_2. Let us use the additional return values to verify that fact.

```
> pos  := P[1]‘ChevBasData‘BasisPos;
> neg  := P[1]‘ChevBasData‘BasisNeg;
> cart := P[1]‘ChevBasData‘BasisCart;
> IsChevalleyBasis(M, RootDatum("G2"), pos, neg, cart);
```

```
true [ <1, 2, 0>, <1, 3, 0>, <1, 4, 0>, <2, 5, 0> ]
```

This demonstrates the fact that the Lie algebra of type G_2 is a constituent of the Lie algebra of type A_3 over fields of characteristic 2.

`RootSystem(L)`

Given a semisimple Lie algebra L with a split Cartan subalgebra, this function computes the root system of L. This function returns four values:

(a) The roots of L with respect to the Cartan subalgebra which is output by `CartanSubalgebra(L)`. This is a sequence of vectors where the positive roots come first, followed by the negative roots.

(b) A sequence of elements of L which are the root vectors corresponding to the roots of L (so the first element corresponds to the first root and so on).

(c) A sequence of simple roots.

(d) The Cartan matrix of the root system with respect to the sequence of simple roots.

Example H106E32

We compute the root system of the simple Lie algebra of type G_2 over the rational field.

```
> L := LieAlgebra("G2", RationalField());
> R, Rv, fund, C:=RootSystem(L);
> R;
[
    (1 0),
    (0 1),
    (1 1),
    (2 1),
    (3 1),
    (3 2),
    (-1  0),
    ( 0 -1),
    (-1 -1),
    (-2 -1),
    (-3 -1),
    (-3 -2)
]
> Rv;
[ (0 0 0 0 0 0 0 0 1 0 0 0 0 0), (0 0 0 0 0 0 0 0 0 0 1 0 0 0 0),
(0 0 0 0 0 0 0 0 0 0 1 0 0 0), (0 0 0 0 0 0 0 0 0 0 0 0 1 0 0),
(0 0 0 0 0 0 0 0 0 0 0 0 1 0), (0 0 0 0 0 0 0 0 0 0 0 0 0 0 1),
(0 0 0 0 0 1 0 0 0 0 0 0 0 0), (0 0 0 0 1 0 0 0 0 0 0 0 0 0),
(0 0 0 1 0 0 0 0 0 0 0 0 0 0), (0 0 1 0 0 0 0 0 0 0 0 0 0 0),
(0 1 0 0 0 0 0 0 0 0 0 0 0 0), (1 0 0 0 0 0 0 0 0 0 0 0 0 0) ]
```

`RootDatum(L)`

Here L is a semisimple Lie algebra. This function returns the root datum D of L with respect to the Cartan subalgebra which is output by `CartanSubalgebra(L)`. We note that the order of the positive roots in D is not necessarily the same as the order in which they appear in the root system of L.

Example H106E33

We set up the root datum of a Lie algebra, and extract the Cartan matrix.

```
> L:= LieAlgebra("F4", Rationals());
> rd := RootDatum(L);
> rd;
Root datum of type F4
> CartanMatrix(rd);
[ 2  0 -1  0]
[ 0  2  0 -1]
[-1  0  2 -1]
[ 0 -1 -2  2]
```

`ChevalleyBasis(L)`

`ChevalleyBasis(L, H)`

`AssumeAlmostSimple` BoolElt *Default :* `false`

Given a semisimple Lie algebra L with a split maximal toral subalgebra H, this function returns three sequences, x, y and h of elements of L. They form a Chevalley basis of L. The first sequence gives basis elements corresponding to positive roots, the second to the negative roots and the third to basis elements in a Cartan subalgebra. If a split maximal toral subalgebra H is not given, an attempt is made to compute one.

For Lie algebras over fields of characteristic 2 and 3 the algorithm used is described in [CR09]. In particular, this involves computing a direct sum decomposition of L, which can be quite time consuming. If there is reason to believe that L is (almost) simple, the optional parameter `AssumeAlmostSimple` should be set to `true`.

Example H106E34

We construct a Chevalley basis for two Lie algebras.

```
> L := LieAlgebra("A2", RationalField());
> x, y, h:= ChevalleyBasis(L);
> x; y; h;
[ (0 0 0 0 0 1 0 0), (0 0 0 0 0 0 1 0), (0 0 0 0 0 0 0 1) ]
[ (0 0 1 0 0 0 0 0), (0 1 0 0 0 0 0 0), (1 0 0 0 0 0 0 0) ]
[ (0 0 0 1 0 0 0 0), (0 0 0 0 1 0 0 0) ]
> L := LieAlgebra("A3", Rationals());
```

```
> print RootDatum(L) : Maximal;
Root datum of type A3 with simple roots
[ 1  0  1]
[ 1 -2  1]
[ 0  1 -2]
and simple coroots
[ 1  1  1]
[ 0 -1  0]
[ 0  0 -1]
```

`ChevalleyBasis(L, H, R)`

Given a semisimple Lie algebra L with a split maximal toral subalgebra H, and an irreducible root datum R, this function computes a Chevalley basis of L with respect to H and R. This basis is returned in the form of three sequences, x, y and h of elements of L, where the first sequence gives basis elements corresponding to positive roots, the second to the negative roots and the third to basis elements in the toral subalgebra H.

`IsChevalleyBasis(L, R, x, y, h)`

Returns `true` if x, y and h form a Chevalley basis of the Lie algebra L with respect to the root datum R. If so, return a sequence describing the extraspecial signs as second return value.

Example H106E35

We compute a Chevalley basis for a Lie algebra of type E_6 inside one of type E_7.

```
> R := RootDatum("E7");
> L1 := LieAlgebra(R, GF(2));
> p1,n1,c1 := StandardBasis(L1);
> L1;
Lie Algebra of dimension 133 with base ring GF(2)
> DynkinDiagram(R);
E7    1 - 3 - 4 - 5 - 6 - 7
              |
              2
> S, proj := sub<R | [1..6]>;
> S;
S: Root datum of dimension 7 of type E6
> #proj;
72
> projpos := [i : i in proj | i le NumPosRoots(R)];
> #projpos;
36
> L2 := sub<L1 | p1[projpos], n1[projpos]>;
> L2;
```

```
Lie Algebra of dimension 78 with base ring GF(2)
> H2 := L2 meet SplitMaximalToralSubalgebra(L1);
> H2;
Lie Algebra of dimension 6 with base ring GF(2)
> p2,n2,c2 := ChevalleyBasis(L2, H2, RootDatum("E6"));
> ok := IsChevalleyBasis(L2, RootDatum("E6"), p2, n2, c2);
> ok;
true
```

`TwistedBasis(L, H, R)`

For a Lie algebra L, a split toral subalgebra H of L, and a twisted root datum R, the function constructs a "twisted basis" of L.

Let k be the coefficient ring of L and K an extension field of k of degree equal to the twisting degree of R. This function has 4 return values. First, $L' = L \otimes K$; second, a homomorphism ϕ from L to L', third, a record containing a Chevalley basis of L' with respect to the untwisted root datum of R; fourth, a matrix describing the action of the Frobenius automorphism of K on the positive roots of the Chevalley basis of L'.

Such a basis constitutes a proof that L' is of type R. Consult [Roo10], Chapter 5.3, for more details on such twisted bases.

Example H106E36

We investigate a twisted basis of the Lie algebra of type ${}^2\mathrm{A}_2$ over the field with 5 elements. Let δ be the automorphism of the root system of type A_2, let k =GF(5), and let K =GF(5^2).

```
> R := TwistedRootDatum(RootDatum("A2") : Twist := 2);
> L := TwistedLieAlgebra(R, GF(5));
> H := SplitToralSubalgebra(L);
> LK, phi, ChevBas, m := TwistedBasis(L, H, R);
> m;
[     0      1]
[     1      0]
```

This matrix `m` shows that δ acts as expected on the Chevalley basis elements of `LK` $= L \otimes K$. We verify the correctness of `m`.

```
> K := CoefficientRing(LK);
> simp := ChevBas`BasisPos[[1..Rank(R)]];
> simp;
[ (     0      0      0      0      0      1  ksi^8      0),
  (     0      0      0      0      0      1 ksi^16      0) ]
> fr := FrobeniusMap(K);
> frv := func<x | Vector([ fr(i) : i in Eltseq(x)])>;
> [ Position(simp, frv(x)) : x in simp ];
```

```
[ 2, 1 ]
```

So indeed the Frobenius map (acting on the coordinates of LK) acts as δ. This is equivalent [Roo10, Lemma 5.3] to the basis elements of L being stable under the composition of the Frobenius map (this time acting on the Chevalley basis of $L \otimes K$) and the root system automorphism δ. We verify this assertion explicitly for this example.

```
> p := ChevBas‘BasisPos;
> n := ChevBas‘BasisNeg;
> c := ChevBas‘BasisCart;
> pi := Sym(6)!(1, 2)(4, 5);
> ChevBasLK := VectorSpaceWithBasis([ Vector(x) : x in p cat n cat c]);
> piL := DiagramAutomorphism(LK, pi);
```

Now δ acts on $L \otimes K$ as `T`, and `fr` is still the Frobenius automorphism of the field K. The images of the basis elements of L under `delta` composed with `fr` are as follows:

```
> for i in [1..Dimension(L)] do
>    b := phi(L.i);
>    printf "i = %o, b =    %o\n", i, Coordinates(ChevBasLK, Vector(b));
>    printf "   pi(b)^fr = %o\n", [ fr(i) : i in
>                              Coordinates(ChevBasLK, Vector(piL(b))) ];
> end for;
i = 1, b =    [ 0, 0, 0, 0, 0, ksi^9, 0, 0 ]
   (b*T)^fr = [ 0, 0, 0, 0, 0, ksi^9, 0, 0 ]
i = 2, b =    [ 0, 0, 0, ksi^5, ksi, 0, 0, 0 ]
   (b*T)^fr = [ 0, 0, 0, ksi^5, ksi, 0, 0, 0 ]
i = 3, b =    [ 0, 0, 0, ksi^9, ksi^21, 0, 0, 0 ]
   (b*T)^fr = [ 0, 0, 0, ksi^9, ksi^21, 0, 0, 0 ]
i = 4, b =    [ 0, 0, 0, 0, 0, 0, ksi^5, ksi ]
   (b*T)^fr = [ 0, 0, 0, 0, 0, 0, ksi^5, ksi ]
i = 5, b =    [ 0, 0, 0, 0, 0, 0, ksi, ksi^5 ]
   (b*T)^fr = [ 0, 0, 0, 0, 0, 0, ksi, ksi^5 ]
i = 6, b =    [ ksi, ksi^5, 0, 0, 0, 0, 0, 0 ]
   (b*T)^fr = [ ksi, ksi^5, 0, 0, 0, 0, 0, 0 ]
i = 7, b =    [ ksi^21, ksi^9, 0, 0, 0, 0, 0, 0 ]
   (b*T)^fr = [ ksi^21, ksi^9, 0, 0, 0, 0, 0, 0 ]
i = 8, b =    [ 0, 0, ksi^9, 0, 0, 0, 0, 0 ]
   (b*T)^fr = [ 0, 0, ksi^9, 0, 0, 0, 0, 0 ]
```

Thus, all the basis elements of L are stable under the composition of the diagram automorphism δ and the Frobenius automorphism.

The `WeylGroup` functions are only available for structure constant Lie algebras.

`WeylGroup(L)`

`WeylGroup(GrpPermCox, L)`

The Weyl group of the reductive Lie algebra L, as a permutation Coxeter group (see Chapter 104).

`WeylGroup(GrpFPCox, L)`

The Weyl group of the reductive Lie algebra L, as a Coxeter group (see Chapter 104).

`WeylGroup(GrpMat, L)`

The Weyl group of the reductive Lie algebra L, as a reflection group (see Chapter 104).

106.9 Operations on Subalgebras and Ideals

`DirectSum(L, M)`

Given Lie algebras L and M, this intrinsic constructs a Lie algebra of dimension $n + m$, where n and m are the dimensions of L and M, respectively. The basis of the new algebra is the concatenation of the bases of L and M and the products $a * b$ where $a \in L$ and $b \in M$ are defined to be zero.

`IndecomposableSummands(L)`

`DirectSumDecomposition(L)`

Given a Lie algebra L, the function returns the direct sum decomposition of L as a sequence of ideals of L whose sum is L and each of which cannot be further decomposed into a direct sum of ideals.

The algorithms used for this function are described in [dG00], §4.12 (semisimple case), §1.15 (general case).

Example H106E37

We compute the direct sum decomposition of the simple Lie algebra of type D_2 over the rational field.

```
> L := LieAlgebra("D2", RationalField());
> L;
Lie Algebra of dimension 6 with base ring Rational Field
> D := DirectSumDecomposition(L);
> D;
[
    Lie Algebra of dimension 3 with base ring Rational Field,
    Lie Algebra of dimension 3 with base ring Rational Field
]
> Morphism(D[1], L);
[ 0  1  0  0  0  0]
[ 0  0  1 -1  0  0]
[ 0  0  0  0  1  0]
> Morphism(D[2], L);
[1 0 0 0 0 0]
[0 0 1 1 0 0]
[0 0 0 0 0 1]
```

106.9.1 Standard Ideals and Subalgebras

`Centre(L)`
`Center(L)`

Given a Lie algebra L, returns the centre of L.

`Centraliser(L, K)`
`Centralizer(L, K)`

Given a Lie algebra L and a subalgebra K of L, returns the centraliser of K in L, and its injection into L.

`Centraliser(L, x)`
`Centralizer(L, x)`

Given a Lie algebra L and an element x of L, returns the centraliser of x in L, and its injection into L.

`Normaliser(L, K)`
`Normalizer(L, K)`

Given a Lie algebra L and a subalgebra K of L, returns the normaliser of K in L, and its injection into L.

`SolubleRadical(L)`
`SolvableRadical(L)`

Given a Lie algebra L, returns the soluble radical of L.

We refer to [dG00], §2.6 for the algorithm used to implement this function.

`Nilradical(L)`

Given a Lie algebra L, returns the nilradical of L.

The algorithm makes use of Cartan subalgebras. We refer to [dG00], pp. 84, 85 for its description.

Example H106E38

We demonstrate the functions for performing basic operations with Lie algebras such as centre, normalizer etc.

```
> L := LieAlgebra("D4", RationalField());
> L;
Lie Algebra of dimension 28 with base ring Rational Field
> Centre(L);
Lie Algebra of dimension 0 with base ring Rational Field
> K := sub< L | [L.1, L.2, L.3] >;
> Centralizer(L, K);
Lie Algebra of dimension 10 with base ring Rational Field
> Normalizer(L, K);
```

```
Lie Algebra of dimension 19 with base ring Rational Field
> M := Centralizer(L, K);
> S := SolvableRadical(M);
> S;
Lie Algebra of dimension 10 with base ring Rational Field
> Morphism(S, L);
[1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
[0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
[0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
[0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
[0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
[0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 1 0 0 0 0 -1 0 0 0 -1 0 0 -1 0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0]
> Nilradical(M);
Lie Algebra of dimension 9 with base ring Rational Field
```

106.9.2 Cartan and Toral Subalgebras

`CartanSubalgebra(L)`

Given a Lie algebra L, this function returns a Cartan subalgebra of L. The algorithm works for Lie algebras L defined over a field F such that $|F| > \dim L$ and for restricted Lie algebras of characteristic p. If the Lie algebra does not fit into one of these classes then the correctness of the output is not guaranteed.

The algorithm used is described in [dG00], §3.2.

`IsCartanSubalgebra(L, H)`

The intrinsic returns `true` if H is a Cartan subalgebra of L, i.e., whether H is nilpotent and $N_L(H) = 0$.

Example H106E39

We compute a Cartan subalgebra of the simple Lie algebra of type A_4 over the rational field.

```
> L := LieAlgebra("F4", RationalField());
> L;
Lie Algebra of dimension 52 with base ring Rational Field
> H := CartanSubalgebra(L);
Lie Algebra of dimension 4 with base ring Rational Field
> H*H;
Lie Algebra of dimension 0 with base ring Rational Field
> Normalizer(L, H);
Lie Algebra of dimension 4 with base ring Rational Field
```

`SplittingCartanSubalgebra(L)`

`SplitMaximalToralSubalgebra(L)`

Given a Lie algebra L over a field k of characteristic at least 5, a split Cartan subalgebra (equivalently, a split maximal toral subalgebra) is computed for L.

The algorithm used is discussed in [CM09], Sections 5 and 6. This algorithm is proved to work ([CM09, Theorem 6.7]) if L is the Lie algebra of a k-split connected reductive group. In other cases, should the algorithm terminate, the output is guaranteed to be correct.

`IsSplittingCartanSubalgebra(L, H)`

Determine whether H is a splitting Cartan subalgebra of L, i.e., whether H is a Cartan subalgebra and the adjoint action of H on L splits completely over the coefficient ring of L.

`SplitToralSubalgebra(L)`

`TryMaximal` . *Default :* `true`

The intrinsic attempts to compute a split toral subalgebra of a Lie algebra L defined over a finite field k. This procedure uses a heuristic algorithm, described in [Roo10, Chapter 3], that works in many cases even if the characteristic of k is small. Moreover, it attempts to compute a split toral subalgebra of maximal size.

If the function returns without error, the resulting subalgebra H is a split toral subalgebra that does not lie inside a split toral subalgebra H' of larger dimension. It is, however, not guaranteed that H is of maximal dimension among all split toral subalgebras.

The optional parameter `TryMaximal` may be used as follows. If set to `true` (the default) the reductive rank r of L is computed first, and the algorithm attempts to compute a split toral subalgebra of dimension r. If set to `false`, the first split toral subalgebra found is returned. Finally, if `TryMaximal` is set to an integer $n \geq 1$, the algorithm attempts to find a split toral subalgebra of dimension n. In the latter case, if no split toral subalgebra of dimension n can be found, the biggest that has been found is returned; if on the other hand a split toral subalgebra of dimension *larger* than n is encountered, that is returned.

`IsSplitToralSubalgebra(L, H)`

Given a restrictable Lie algebra L over a finite field, the function returns `true` is H is a split toral subalgebra of L, i.e., whether $[H, H] = 0$, all elements of H are semisimple, and the basis elements are invariant under the q-map associated to L.

Example H106E40

We construct a twisted Lie algebra L of type ${}^3\mathrm{D}_4$ over the field $k = \mathrm{GF}(3^3)$ and verify that the subalgebra H returned by `SplitToralSubalgebra` is indeed a split toral subalgebra. Then, we test whether $C = C_L(H)$ is a (split) toral subalgebra of L.

```
> k := GF(3, 3);
```

```
> L, phi := TwistedLieAlgebra(TwistedRootDatum("D4" : Twist := 3), k);
> H := SplitToralSubalgebra(L);
> H;
Lie Algebra of dimension 2 with base ring GF(3^3)
> IsSplitToralSubalgebra(L, H);
true
> C := Centraliser(L,H); C;
Lie Algebra of dimension 4 with base ring GF(3^3)
> IsToralSubalgebra(L,C), IsSplitToralSubalgebra(L, C);
true false
```

Now we let K be the big field, $\mathrm{GF}(3^9)$, and test if $C \otimes K$ is a split toral subalgebra of $L \otimes K$.

```
> LK := Codomain(phi);
> LK;
Lie Algebra of dimension 28 with base ring GF(3^9)
> CK := sub<LK | [ phi(b) : b in Basis(C) ]>;
> IsSplitToralSubalgebra(LK, CK);
true
```

106.9.3 Standard Series

`CompositionSeries(L)`

A composition series is computed for the (structure constant) Lie algebra L. The function returns three values:

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

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

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

`CompositionFactors(L)`

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

`MinimalIdeals(L : parameters)`

`Limit` RngIntElt *Default :* ∞

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

`MaximalIdeals(L : parameters)`

`Limit` RngIntElt *Default :* ∞

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

`DerivedSeries(L)`

Given a Lie algebra L, this function returns a sequence of ideals of L that form its derived series.

`LowerCentralSeries(L)`

Given a Lie algebra L, this function returns a sequence of ideals of L that form its lower central series.

`UpperCentralSeries(L)`

Given a Lie algebra L, this function returns a sequence of ideals of L that form the upper central series of L. The function repeatedly uses the algorithm for computing centres while keeping track of the pre-images of the ideals factored out.

Example H106E41

We compute each of the type of series of a particular subalgebra of the simple Lie algebra of type F_4 over the rational field.

```
> L:=LieAlgebra("F4", RationalField());
> L;
Lie Algebra of dimension 52 with base ring Rational Field
> K:=sub< L | [L.1, L.12, L.23, L.34, L.45] >;
> DerivedSeries(K);
[
    Lie Algebra of dimension 20 with base ring Rational Field,
    Lie Algebra of dimension 16 with base ring Rational Field,
    Lie Algebra of dimension 7 with base ring Rational Field,
    Lie Algebra of dimension 0 with base ring Rational Field
]
> LowerCentralSeries(K);
[
    Lie Algebra of dimension 20 with base ring Rational Field,
    Lie Algebra of dimension 16 with base ring Rational Field,
    Lie Algebra of dimension 12 with base ring Rational Field,
    Lie Algebra of dimension 8 with base ring Rational Field,
    Lie Algebra of dimension 5 with base ring Rational Field,
    Lie Algebra of dimension 2 with base ring Rational Field,
    Lie Algebra of dimension 1 with base ring Rational Field,
    Lie Algebra of dimension 0 with base ring Rational Field
]
> UpperCentralSeries(K);
```

```
[
    Lie Algebra of dimension 2 with base ring Rational Field,
    Lie Algebra of dimension 3 with base ring Rational Field,
    Lie Algebra of dimension 5 with base ring Rational Field,
    Lie Algebra of dimension 8 with base ring Rational Field,
    Lie Algebra of dimension 12 with base ring Rational Field,
    Lie Algebra of dimension 16 with base ring Rational Field,
    Lie Algebra of dimension 20 with base ring Rational Field
]
```

106.9.4 The Lie Algebra of Derivations

`LieAlgebraOfDerivations(L)`

Given a Lie algebra L, this function constructs its Lie algebra of derivations $\mathrm{Der}(L)$. As second return value, a record containing maps from L to $\mathrm{Der}(L)$ and vice versa, and from $\mathrm{Der}(L)$ to the matrix Lie algebra acting on L is returned.

Example H106E42

We consider the Lie algebra of derivations of D_4 in characteristic 2 or, more precisely, the 26-dimensional simple constituent L that exists in all varieties of D_4 in characteristic 2.

```
> SetSeed(1);
> R := RootDatum("D4");
> D4 := LieAlgebra(R, GF(2));
> pos,neg,cart := StandardBasis(D4);
> L := D4*D4; L;
Lie Algebra of dimension 26 with base ring GF(2)
> IsSimple(L);
true
> DerL, maps := LieAlgebraOfDerivations(L);
> DerL;
Lie Algebra of dimension 52 with base ring GF(2)
> SemisimpleType(DerL);
F4
```

So the Lie algebra of derivations is of type F_4. Let us consider one of the maps that was returned as second value.

```
> maps;
rec<recformat<mp_DerL_to_L: Map, mp_L_to_DerL: Map, mp_DerL_to_mats:
Map, mp_mats_to_DerL: Map> |
    mp_DerL_to_L := Mapping from: AlgLie: DerL to AlgLie: L given by a
        rule [no inverse],
    mp_L_to_DerL := Mapping from: AlgLie: L to AlgLie: DerL given by a
        rule [no inverse],
    mp_DerL_to_mats := Mapping from: AlgLie: DerL to Matrix Lie
```

```
        Algebra given by a rule [no inverse],
    mp_mats_to_DerL := Mapping from: Matrix Lie Algebra to AlgLie:
        DerL given by a rule [no inverse]>
> adL := AdjointRepresentation(L);
> f := maps‘mp_DerL_to_mats;
> [ f(b) in Image(adL) : b in Basis(DerL) ];
[ false, true, true, true, true, true, true, true, false, false, true,
false, true, false, false, false, true, false, true, true, true,
false, false, false, true, true, false, false, false, true, true,
false, false, false, true, true, false, false, true, false, true,
false, true, false, false, false, true, false, true, false, false,
false ]
```

So, unsurprisingly, some of the basis elements of $\mathrm{Der}(L)$ are actually elements from L, but others are not. We consider one more of these maps and investigate how L lies in $\mathrm{Der}(L)$.

```
> g := maps‘mp_L_to_DerL;
> I := ideal<DerL | [ g(b) : b in Basis(L) ]>; I;
Lie Algebra of dimension 26 with base ring GF(2)
> pos2, neg2, cart2 := ChevalleyBasis(DerL, SplitToralSubalgebra(DerL));
> [i : i in [1..#pos2] | pos2[i] in I ];
[ 3, 4, 6, 7, 8, 10, 12, 13, 15, 17, 19, 21 ]
> RF4 := RootDatum("F4");
> [ i : i in [1..NumPosRoots(RF4)] | IsShortRoot(RF4, i) ];
[ 3, 4, 6, 7, 8, 10, 12, 13, 15, 17, 19, 21 ]
```

So we conclude that the original Lie algebra L of type D_4 exists as the short roots of the Lie algebra of derivations $\mathrm{Der}(L)$ of type F_4.

106.10 Properties of Lie Algebras and Ideals

`KillingMatrix(L)`

Given a Lie algebra L such that $\{x_1, \ldots, x_n\}$ is a basis of L, return the Killing matrix of L, which is defined to be the matrix $(\mathrm{Tr}(\mathrm{ad}x_i \cdot \mathrm{ad}x_j))$.

Example H106E43

```
> L:=LieAlgebra("B2",RationalField());
> KillingMatrix(L);
[ 0  0  0 -6  0  0  0  0  0  0]
[ 0  0 -6  0  0  0  0  0  0  0]
[ 0 -6  0  0  0  0  0  0  0  0]
[-6  0  0  0  0  0  0  0  0  0]
[ 0  0  0  0  0  0  0  0  0  6]
[ 0  0  0  0  0  0  0  0  6  0]
[ 0  0  0  0  0  0  6  0  0  0]
```

```
[ 0  0  0  0  0  0  0  6  0  0]
[ 0  0  0  0  0  6  0  0  0  0]
[ 0  0  0  0  6  0  0  0  0  0]
```

`IsAbelian(L)`

Given a Lie algebra L, return `true` if L is abelian.

`IsSoluble(L)`
`IsSolvable(L)`

Given a Lie algebra L, return `true` if L is soluble.

`IsNilpotent(L)`

Given a Lie algebra L, return `true` if L is nilpotent.

`IsCentral(L, M)`

Given a subalgebra M of the Lie algebra L, return `true` if M is central in L.

`IsSimple(L)`

Given a Lie algebra L, return `true` if L is simple.

`IsSemisimple(L)`

Given a Lie algebra L, return `true` if L is semisimple.

`IsReductive(L)`

Given a Lie algebra L, return `true` if L is reductive.

`HasLeviSubalgebra(L)`

Given a Lie algebra L, this function determines whether L has a Levi subalgebra. If the result is `true`, then the function also returns a semisimple subalgebra (complement to the solvable radical) of L. If L is defined over a field of characteristic 0, then it always has a Levi subalgebra. However, if L is a Lie algebra of characteristic $p > 0$ then L need not have a Levi subalgebra but the function will always find one if it exists.

A description of the algorithm used is contained in [dG00], §4.13.

`IsClassicalType(L)`

Determines if the reductive Lie algebra L is of classical-type. Note that all reductive Lie algebras over fields of characteristic 0 are considered to be classical-type.

Example H106E44

We test various predicates in the context of the simple Lie algebra of type D_3 over the rational field.

```
> L:=LieAlgebra("D3",RationalField());
> L;
Lie Algebra of dimension 15 with base ring Rational Field
> K:=sub< L | [L.1,L.2,L.3] >;
> M:=Centralizer(L, K);
> M;
Lie Algebra of dimension 4 with base ring Rational Field
> R:=SolvableRadical(M);
> R;
Lie Algebra of dimension 4 with base ring Rational Field
> HasLeviSubalgebra(M);
true Lie Algebra of dimension 0 with base ring Rational Field
> K:=Centralizer(L, sub< L | [L.1,L.2,L.3] >);
> K;
Lie Algebra of dimension 4 with base ring Rational Field
> IsSolvable(K);
true
> IsNilpotent(K);
true
> R:= SolvableRadical(K);
> IsSolvable(R);
true
> IsNilpotent(R);
true
> N:= Nilradical(K);
> IsNilpotent(N);
true
```

106.11 Operations on Elements

`x + y` `x - y` `x * y`

`IsCentral(L, M)`

Given an element x of the Lie algebra L, return `true` if x is central in L.

`NonNilpotentElement(L)`

Given a (structure constant) Lie algebra L, this function returns an element of L that is *not* nilpotent, or the zero element of L if no such element exists.

The algorithm follows [dG00], §2.7.

Example H106E45

We construct a non-nilpotent element of a Lie algebra.

```
> L:=LieAlgebra("G2",RationalField());
> NonNilpotentElement(L);
(0 0 0 0 0 1 0 0 0 0 0 0 0 0)
```

`AdjointMatrix(L, x)`

`RightAdjointMatrix(L, x)`

Given a (structure constant) Lie algebra L and an element x of a subalgebra or ideal of L, return the matrix of $\mathrm{ad}x$ as an element of a matrix Lie algebra.

Example H106E46

```
> L:=LieAlgebra("B2",RationalField());
> AdjointMatrix(L, L.1);
[ 0  0  0  0  0  0  0  0  0  0]
[ 0  0  0  0  0  0  0  0  0  0]
[ 0  0  0  0  0  0  0  0  0  0]
[ 0  0  0  0  0  0  0  0  0  0]
[ 1  0  0  0  0  0  0  0  0  0]
[ 2  0  0  0  0  0  0  0  0  0]
[ 0  0  0  0  0  0  0  0  0  0]
[ 0 -1  0  0  0  0  0  0  0  0]
[ 0  0  1  0  0  0  0  0  0  0]
[ 0  0  0  0  0 -1  0  0  0  0]
```

106.11.1 Indexing

`a[i]`

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

If a is an element of a matrix Lie algebra L of degree n and $1 \leq i \leq n$ then the ith row of the matrix a is returned.

`a[i] := r`

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

If a is an element of a matrix Lie algebra L of degree n over R and $1 \leq i \leq n$, the ith row of the matrix a is redefined to be the vector r over R.

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

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

For an element a of a matrix Lie algebra L of degree n and integers $1 \leq i, j \leq n$ return the element in the ith row and jth column of a or set this element to be r where r is an element of the coefficient ring of L.

106.12 The Natural Module

```
Module(L)
```

The module R^n underlying the Lie algebra L.

```
RModule(L)
```

The module R^n acted on by the matrix Lie algebra L.

```
BaseModule(L)
```

The space R^n acted on by the matrix Lie algebra L.

```
Degree(L)
```

The degree of the Lie algebra L. If L is a structure constant algebra, this is just the dimension of L. If L is a matrix Lie algebra, this is the degree of the matrices in L.

```
Degree(a)
```

Given an element a belonging to the Lie algebra L, the dimension of L is returned.

```
ElementToSequence(a)
```

```
Eltseq(a)
```

The sequence of coefficients of the Lie element a.

```
Coordinates(M, a)
```

Let a be an element of a Lie algebra L and let M be a subalgebra of L containing a. This function returns the coefficients of a with respect to the basis of L.

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

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

```
Support(a)
```

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

106.13 Operations for Matrix Lie Algebras

This section describes the functionality provided for matrix Lie algebras which is additional to that provided for structure constant Lie algebras. For further information see Chapter 88.

`BaseModule(M)`

The natural module on which the matrix Lie algebra M acts.

`Generic(M)`

The full matrix algebra in which the matrix Lie algebra M is naturally embedded.

`Kernel(X)`
`Nullspace(X)`

The kernel of the homomorphism represented by the Lie matrix algebra element X.

`NullspaceOfTranspose(X)`
`RowNullSpace(X)`

The row nullspace of the homomorphism represented by the Lie matrix algebra element X.

106.14 Homomorphisms

`hom< L -> M | Q >`

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

Note that this is in general only a module homomorphism, and no check is made for it being an algebra homomorphism.

106.15 Automorphisms of Classical-type Reductive Algebras

`IdentityAutomorphism(L)`

The trivial automorphism of the Lie algebra L.

`InnerAutomorphism(L, x)`

The inner automorphism of the Lie algebra L induced by x, where x is an element of the corresponding group of Lie type.

`InnerAutomorphismGroup(L)`

The group of Lie type G corresponding to the Lie algebra L. The map $G \to \mathrm{Aut}(L)$ is returned as second value.

`DiagonalAutomorphism(L, v)`

The diagonal automorphism of the Lie algebra L induced by the vector v.

`GraphAutomorphism(L, p)`

`DiagramAutomorphism(L, p)`

`SimpleSigns` ANY *Default :* 1

The graph automorphism of the Lie algebra L induced by the permutation p. This must be either a permutation of the indices of the simple roots, or a permutation of the indices of all roots.

The optional parameter `SimpleSigns` can be used to specify the signs corresponding to each simple root. This should either be a sequence of integers ± 1, or a single integer ± 1.

Example H106E47

We construct an automorphism of order three for the simple Lie algebra of type D_4.

```
> DynkinDiagram( "D4" );
D4    3
     /
1 - 2
     \
      4
> p:= Sym(4)!(1,3,4);
> L:= LieAlgebra( "D4", Rationals() );
> f:= GraphAutomorphism( L, p );
> f(L.3);
(0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0)
> f(L.4);
(0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0)
> f(L.5);
(0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0)
```

106.16 Restrictable Lie Algebras

A restricted Lie algebra is a Lie algebra over a field of characteristic $p > 0$, equipped with a restriction map $x \to x^p$, satisfying the axioms given in [Jac62]. A restrictable Lie algebra is a Lie algebra which can be equipped with a restriction map. A Lie algebra is restrictable if and only if ad$L$ is closed under the pth power map. Hence restrictable Lie algebras have a standard restriction map induced by the adjoint representation. For many purposes, it suffices to know that a Lie algebra is restrictable, without needing to know a restriction.

By convention, a Lie algebra over a field of characteristic zero is always considered restrictable, and the restriction map is the identity map.

In MAGMA, we do not make a distinction between the concepts of restricted and restrictable. Note however that a Lie algebra can have a nonstandard restriction map.

`IsRestrictable(L)`
`IsRestricted(L)`
`IspLieAlgebra(L)`

Returns `true` if, and only if, the Lie algebra L is restrictable. If L is restrictable, the restriction map is returned as a second value.

`RestrictionMap(L)`
`pMap(L)`

The restriction map of the Lie algebra L. If L is not restrictable, an error is signalled.

Example H106E48

```
> L:= LieAlgebra( "A2", GF(5) );
> IsRestrictable( L );
true Mapping from: AlgLie: L to AlgLie: L given by a rule [no inverse]
> pmap:= pMap( L );
> pmap( 2*L.3 + L.4);
(0 0 0 1 0 0 0 0)
```

`RestrictedSubalgebra(Q)`
`pSubalgebra(Q)`

Given a sequence Q of elements from the Lie algebra L, the function returns the restricted subalgebra generated by the elements of Q, i.e., the smallest subalgebra containing Q which is also closed under the restriction map. If the parent of Q is not restrictable, an error is signalled.

`pClosure(L, M)`

Given Lie algebras L and M such that $L \leq M$, this function returns the closure of L under the restriction map of M. If L is not a subalgebra of M or M is not restrictable, an error is signalled.

`IsRestrictedSubalgebra(L, M)`

`IspSubalgebra(L, M)`

Return `true` if and only if the Lie algebra L is a restricted Lie subalgebra of M with the same restriction map. Note that if L is constructed using the `pClosure` intrinsic, this will always be true. However if L is constructed as a subalgebra, this may be false even if L is restrictable, since the restriction map of L will be the standard map rather than the restriction map of M.

`pQuotient(L, M)`

Given Lie algebras L and M such that $L \leq M$, this function returns the quotient of L by the p-closure of the Lie algebra M, with respect to the inherited restriction map.

`JenningsLieAlgebra(G)`

Let G be a p-group. Then the quotients of the successive terms of the Jennings series of G can be viewed as vector spaces over the field of p elements. The direct sum of these vector spaces carries the structure of a Lie algebra (coming from the commutator of G). This function returns two values. Firstly, the Lie algebra constructed from G by this process. This Lie algebra is graded. The second returned value is a sequence of sequences of two elements. The first element is the degree of a homogeneous component while the second element is its dimension. The basis elements of the Lie algebra are ordered according to increasing degree. This means that from the dimensions of the homogeneous components it is possible to derive the degree of each basis element.

Lie algebras constructed in this way are naturally restricted. Moreover, if x is a homogeneous element of degree d, then the p-th power image of x is homogeneous of degree pd.

Example H106E49

```
> G:= SmallGroup( 3^6, 196 );
> L, gr:= JenningsLieAlgebra( G );
> L;
Lie Algebra of dimension 6 with base ring GF(3)
> gr;
[
    [ 1, 3 ],
    [ 2, 1 ],
    [ 3, 2 ]
```

```
]
// So the first three basis elements are of degree 1,
// the fourth basis element is of degree 2, and so on.
> pmap:= pMap( L );
> pmap( L.1 );
(0 0 0 0 1 1)
```

106.17 Universal Enveloping Algebras

This section describes the functionality for universal enveloping algebras of Lie algebras. If a Lie algebra is semisimple and defined over a field of characteristic 0, then it is possible to write down an integral basis of the universal enveloping algebra that has nice properties. To accommodate this possibility, two constructions of a universal enveloping algebra are provided: a general construction, and one in which this integral basis is used. First we briefly describe the theoretical background behind universal enveloping algebras.

In MAGMA, universal enveloping algebras have type `AlgUE` and their elements have type `AlgUEElt`. Integral universal enveloping algebras have type `AlgIUE` and their elements have type `AlgIUEElt`. General algebras having a PBW basis (see below) have type `AlgPBW` (elements type `AlgPBWElt`) which inherit from types `Alg` and `Rng`. Consequently, the type `AlgIUE` inherits from `AlgPBW`.

106.17.1 Background

106.17.1.1 Universal Enveloping Algebras

Let L be a Lie algebra over the field F having basis $x_1, \ldots, x_n$. The universal enveloping algebra $U(L)$ of L is the associative algebra with identity, generated by n symbols which are also denoted by $x_1, \ldots, x_n$. These generators satisfy the relations

$$x_j x_i - x_i x_j = [x_j, x_i], 1 \leq i, j \leq n > .$$

Here $[x_j, x_i]$ is the product in the Lie algebra L, so it is a certain linear combination of the x_k.

The theorem of Poincaré-Birkhoff-Witt states that a basis of $U(L)$ is formed by the set of all elements

$$x_1^{k_1} \cdots x_n^{k_n},$$

where the k_i are non-negative integers. Furthermore, the product of two such basis elements may be rewritten as a linear combination of basis elements using the defining relations $x_j x_i - x_i x_j = [x_j, x_i]$ for $j > i$.

106.17.1.2 The Integral Form of a Universal Enveloping Algebra

If the Lie algebra L happens to be (split) semisimple and of characteristic 0, then the universal enveloping algebra has a nice basis described by [Kos66]. The first step in constructing this basis involves taking a Chevalley basis of L, consisting of the elements $y_1, \ldots, y_s$, $h_1, \ldots, h_r$ and $x_1, \ldots, x_s$. Here the y_i and x_i are root vectors belonging to negative roots and positive roots, respectively. The h_i are basis elements of a Cartan subalgebra. In the universal enveloping algebra we use the divided powers

$$y_i^{(n)} = \frac{y_i^n}{n!}, \quad x_i^{(n)} = \frac{x_i^n}{n!},$$

and the binomials

$$\binom{h_i}{k} = \frac{h_i(h_i-1)\cdots(h_i-k+1)}{k!}.$$

A basis of $U(L)$ is formed by the elements

$$y_1^{(m_1)} \cdots y_s^{(m_s)} \binom{h_1}{k_1} \cdots \binom{h_r}{k_r} x_1^{(n_1)} \cdots x_s^{(n_s)}.$$

This basis has the useful property that if we multiply two basis elements, the structure constants will be integers (usually of quite moderate size). So this is a basis of an integral form of the universal enveloping algebra.

106.17.2 Construction of Universal Enveloping Algebras

`UniversalEnvelopingAlgebra(L)`

This creates the universal enveloping algebra U of the Lie algebra L. Here the i-th basis element of L (i.e., `L.i`) corresponds to the i-th generator of U (i.e., `U.i`). Every product of generators is rewritten as a linear combination of Poincaré-Birkhoff-Witt monomials (cf. Section 106.17.1.1).

`IntegralUEA(L)`

`IntegralUEAlgebra(L)`

`IntegralUniversalEnvelopingAlgebra(L)`

Given a semisimple Lie algebra L of characteristic 0, create the integral universal enveloping algebra U of L. The basis described in Section 106.17.1.2 is used.

Let x, y and h denote the output of `ChevalleyBasis(L)`. Let s be the length of x, and r the length of h. Then every generator of U corresponds to an element of x, y or h. If $1 \leq i \leq s$ then the i-th generator of U (i.e., `U.i`) corresponds to the i-th element of y. It is printed as `y_i`. If $s+1 \leq i \leq s+r$, then the i-th generator of U corresponds to the k-th element of h, where $k = i - s$. It is printed as `[ h_k ; 1 ]` (i.e., `h_k` choose 1). Finally, if $s+r+1 \leq i \leq 2s+r$, then the i-th generator corresponds to the k-th element of x, where $k = i - s - r$. It is printed as `x_k`.

Using this form of the universal enveloping algebra has two advantages. Firstly, the structure constants are integers which usually remain relatively small. Secondly, multiplication of elements is, in general, much faster than is the case with universal enveloping algebras that employ PBW bases.

Example H106E50

```
> T:= [ <4,1,1,1>, <1,4,1,-1>, <4,1,3,1>, <1,4,3,-1>, <4,2,2,1>, <2,4,2,-1>,
> <4,3,1,1>, <3,4,1,-1>, <3,1,2,1>, <1,3,2,-1> ];
> L:= LieAlgebra< Rationals(), 4 | T >;
> U:= UniversalEnvelopingAlgebra(L);
> U.4*U.1;
x_1*x_4 + x_1 + x_3
> L:= LieAlgebra("F4", Rationals());
> U:= IntegralUEA(L);
> U.29*U.1;
y_1*x_1 + [ h_1 ; 1 ]
> (1/4)*U.29^2*U.1^2;
y_1^(2)*x_1^(2) + y_1*[ h_1 ; 1 ]*x_1 - 2*y_1*x_1 + [ h_1 ; 2 ]
```

In the last example we divided by 4 because `U.29^2 = 2 U.29^(2)`, and likewise for `U.1^2`.

`AssignNames(~U, Q)`

Assign the names in the sequence Q to the generators of the algebra U.

`ChangeRing(U, S)`

Given a universal enveloping algebra U with base ring R, together with a ring S, construct the algebra U' with base ring S obtained by coercing the coefficients of elements of U into S.

106.17.3 Related Structures

`CoefficientRing(U)`

`BaseRing(U)`

The ring of coefficients of the universal enveloping algebra U.

`Algebra(U)`

The Lie algebra corresponding to the universal enveloping algebra U.

106.17.4 Elements of Universal Enveloping Algebras

Most functions in this section are applicable both to universal enveloping algebras and to integral universal enveloping algebras. Therefore, they are only documented once. An exception is the function `HBinomial`, which is only applicable to integral universal enveloping algebras.

106.17.4.1 Creation of Elements

`U ! 0`

`Zero(U)`

The zero element of the universal enveloping algebra U.

`U ! 1`

`One(U)`

The identity element of the universal enveloping algebra U.

`U . i`

The i-th generator of the universal enveloping algebra U.

`U ! r`

Returns r as an element of the enveloping algebra U where r may be anything coercible into the coefficient ring of U or an element of another enveloping algebra of the same type as U whose coefficients can be coerced into the coefficient ring of U.

`HBinomial(U, i, n)`

`HBinomial(h, n)`

This function is applicable only in the case of integral universal enveloping algebras. It is used for constructing the "binomial" elements h_i choose n. In the first form U is an integral universal enveloping algebra, and i is an index between 1 and the rank of the root datum. In the second form, the element h is simply `U.(s+i)`, where s is the number of positive roots.

Example H106E51

```
> L:= LieAlgebra("E6",Rationals());
> U:= IntegralUEA(L);
> HBinomial(U, 4, 10);
[ h_4 ; 10 ]
```

106.17.4.2 Operations on Elements

`x + y` `x - y` `x * y` `c * x` `x * c` `x ^ n`

`Monomials(u)`

The sequence of the monomials that occur in the element u of a universal enveloping algebra.

`Coefficients(u)`

The sequence of coefficients of the monomials in the element u of a universal enveloping algebra. The k-th element of this sequence corresponds exactly to the k-th monomial in the sequence returned by `Monomials(u)`.

`Degree(u, i)`

Given an element u of a universal enveloping algebra U and an integer i, this function returns the degree of u in the i-th generator of U.

Example H106E52

```
> L:= LieAlgebra("G2",Rationals());
> U:= IntegralUEA(L);
> c:= U.7*U.2;c;
y_2*[ h_1 ; 1 ] + 3*y_2
> Monomials(c);
[
    y_2*[ h_1 ; 1 ],
    y_2
]
> Coefficients(c);
[ 1, 3 ]
> c:= U.10*U.7*U.2; c;
y_2*[ h_1 ; 1 ]*x_2 + 6*y_2*x_2 + [ h_1 ; 1 ]*[ h_2 ; 1 ] + 3*[ h_2 ; 1 ]
> Degree(c, 2);
1
> Degree(c, 7);
1
> Degree(c, 8);
1
```

106.18 Solvable and Nilpotent Lie Algebras Classification

This section describes functions for working with the classification of solvable Lie algebras of dimension 2, 3, and 4, and the classification of nilpotent Lie algebras having dimensions 3,4,5, and 6. The classification of solvable Lie algebras is taken from [dG05], and applies to algebras over any base field. The classification of nilpotent Lie algebras is taken from [dG07]. It lists the nilpotent Lie algebras over any base field, with the exception of fields of characteristic 2, when the dimension is 6.

The functions described here fall into two categories: functions for creating the Lie algebras of the classification, and a function for identifying a given solvable Lie algebra of dimension 2,3,4 or a given nilpotent Lie algebra of dimension 3,4,5,6 as a member of the list.

First we describe the classifications, in order to define names for the Lie algebras that occur. We then describe the functions for working with them in MAGMA.

106.18.1 The List of Solvable Lie Algebras

We denote a solvable Lie algebra of dimension n by L_n^k, where k ranges between 1 and the number of classes of solvable Lie algebras of dimension n. If the class depends on a parameter, say a, then we denote the Lie algebra by $L_n^k(a)$. In such cases we also state conditions under which $L_n^k(a)$ is isomorphic to $L_n^k(b)$ (if there are any). We list the nonzero commutators only. The field over which the Lie algebra is defined is denoted by F. Here is the list of classes of solvable Lie algebras having dimension not greater than 4:

L_2^1 Abelian.

L_2^2 $[x_2, x_1] = x_1$.

L_3^1 Abelian.

L_3^2 $[x_3, x_1] = x_1$, $[x_3, x_2] = x_2$.

$L_3^3(a)$ $[x_3, x_1] = x_2$, $[x_3, x_2] = ax_1 + x_2$.

$L_3^4(a)$ $[x_3, x_1] = x_2$, $[x_3, x_2] = ax_1$. Condition of isomorphism: $L_3^4(a) \cong L_3^4(b)$ if and only if there is an $\alpha \in F^*$ with $a = \alpha^2 b$.

L_4^1 Abelian.

L_4^2 $[x_4, x_1] = x_1$, $[x_4, x_2] = x_2$, $[x_4, x_3] = x_3$.

$L_4^3(a)$ $[x_4, x_1] = x_1$, $[x_4, x_2] = x_3$, $[x_4, x_3] = -ax_2 + (a+1)x_3$.

L_4^4 $[x_4, x_2] = x_3$, $[x_4, x_3] = x_3$.

L_4^5 $[x_4, x_2] = x_3$.

$L_4^6(a, b)$ $[x_4, x_1] = x_2$, $[x_4, x_2] = x_3$, $[x_4, x_3] = ax_1 + bx_2 + x_3$.

$L_4^7(a, b)$ $[x_4, x_1] = x_2$, $[x_4, x_2] = x_3$, $[x_4, x_3] = ax_1 + bx_2$. Isomorphism condition: $L_4^7(a, b) \cong L_4^7(c, d)$ if and only if there is an $\alpha \in F^*$ with $a = \alpha^3 c$ and $b = \alpha^2 d$.

L_4^8 $[x_1, x_2] = x_2$, $[x_3, x_4] = x_4$.

$L_4^9(a)$ $[x_4, x_1] = x_1 + ax_2$, $[x_4, x_2] = x_1$, $[x_3, x_1] = x_1$, $[x_3, x_2] = x_2$. Condition on the parameter a: $T^2 - T - a$ has no roots in F. Isomorphism condition: $L_4^9(a) \cong L_4^9(b)$ if and only if the characteristic of F is not 2 and there is an $\alpha \in F^*$ with $a + \frac{1}{4} = \alpha^2(b + \frac{1}{4})$, or the characteristic of F is 2 and $X^2 + X + a + b$ has roots in F.

$L_4^{10}(a)$ $[x_4, x_1] = x_2$, $[x_4, x_2] = ax_1$, $[x_3, x_1] = x_1$, $[x_3, x_2] = x_2$. Condition on F: the characteristic of F is 2. Condition on the parameter a: $a \notin F^2$. Isomorphism condition: $L_4^{10}(a) \cong L_4^{10}(b)$ if and only if $Y^2 + X^2 b + a$ has a solution $(X, Y) \in F \times F$ with $X \neq 0$.

$L_4^{11}(a, b)$ $[x_4, x_1] = x_1$, $[x_4, x_2] = bx_2$, $[x_4, x_3] = (1 + b)x_3$, $[x_3, x_1] = x_2$, $[x_3, x_2] = ax_1$. Condition on F: the characteristic of F is 2. Condition on the parameters a, b: $a \neq 0$, $b \neq 1$. Isomorphism condition: $L_4^{11}(a, b) \cong L_4^{11}(c, d)$ if and only if $\frac{a}{c}$ and $(\delta^2 + (b + 1)\delta + b)/c$ are squares in F, where $\delta = (b + 1)/(d + 1)$.

L_4^{12} $[x_4, x_1] = x_1$, $[x_4, x_2] = 2x_2$, $[x_4, x_3] = x_3$, $[x_3, x_1] = x_2$.

$L_4^{13}(a)$ $[x_4, x_1] = x_1 + ax_3$, $[x_4, x_2] = x_2$, $[x_4, x_3] = x_1$, $[x_3, x_1] = x_2$.

$L_4^{14}(a)$ $[x_4, x_1] = ax_3$, $[x_4, x_3] = x_1$, $[x_3, x_1] = x_2$. Condition on parameter a: $a \neq 0$. Isomorphism condition: $L_4^{14}(a) \cong L_4^{14}(b)$ if and only if there is an $\alpha \in F^*$ with $a = \alpha^2 b$.

106.18.2 Comments on the Classification over Finite Fields

Over general fields the lists are not "precise" in the sense that some classes that depend up on a parameter have an associated isomorphism condition, but not a precise parametrization of the Lie algebras in that class. However, for algebras over finite fields we are able to give a precise list, by restricting the parameter values in some cases. In this section we describe how this is done. Here F will be a finite field of size q with primitive root γ.

* If the characteristic of F is 2, then there are two algebras of type $L_3^4(a)$, namely $L_3^4(0)$ and $L_3^4(1)$. If the characteristic is not 2, then there are three algebras of this type, $L_3^4(0)$, $L_3^4(1)$, $L_3^4(\gamma)$.

* The class $L_4^7(a, b)$ splits into three classes: $L_4^7(a, a)$ $(a \in F)$, $L_4^7(a, 0)$ $(a \neq 0)$, $L_4^7(0, b)$ $(b \neq 0)$. Among the algebras of the first class there are no isomorphisms. However, for the other two classes we have the following:-

(i) $L_4^7(a, 0) \cong L_4^7(b, 0)$ if and only if there is an $\alpha \in F^*$ such that $a = \alpha^3 b$. If $q \equiv 1 \bmod 3$, then exactly a third of the elements of F^* are cubes, namely the γ^i with i divisible by 3. So in this case we get three algebras, $L_4^7(1, 0)$, $L_4^7(\gamma, 0)$, $L_4^7(\gamma^2, 0)$. If $q \not\equiv 1 \bmod 3$ then $F^3 = F$, and hence there is only one algebra, namely $L_4^7(1, 0)$.

(ii) $L_4^7(0, a) \cong L_4^7(0, b)$ if and only if there is an $\alpha \in F^*$ such that $a = \alpha^2 b$. So if q is even then we get one algebra, $L_4^7(0, 1)$. If q is odd we get two algebras, $L_4^7(0, 1)$, $L_4^7(0, \gamma)$.

* In [dG05] it is shown that there is only one Lie algebra in the class $L_4^9(a)$. We let e be the smallest positive integer such that $T^2 - T - \gamma^e$ has no roots in F. Then we take the Lie algebra $L_4^9(\gamma^e)$ as representative of the class.
* Over a finite field of characteristic 2 there are no Lie algebras of type $L_4^{10}(a)$, as $F^2 = F$ in that case.
* There is only one Lie algebra of type $L_4^{11}(a, b)$ over a field of characteristic 2, namely $L_4^{11}(1, 0)$.
* If q is even then there is only one algebra of type $L_4^{14}(a)$, namely $L_4^{14}(1)$. If q is odd, then there are two algebras, $L_4^{14}(1)$ and $L_4^{14}(\gamma)$.

106.18.3 The List of Nilpotent Lie Algebras

We denote a nilpotent Lie algebra of dimension r by N_r^k, where k ranges between 1 and the number of classes of nilpotent Lie algebras of dimension r. If the class depends on a parameter, say a, then we denote the Lie algebra by $N_r^k(a)$. The complete list of isomorphism classes of nilpotent Lie algebras having dimensions 3, 4, 5 and 6, *where in dimension 6 we exclude base fields of characteristic 2* are as follows:

N_3^1 Abelian.
N_3^2 $[x_1, x_2] = x_3$.
N_4^1 Abelian.
N_4^2 $[x_1, x_2] = x_3$.
N_4^3 $[x_1, x_2] = x_3$, $[x_1, x_3] = x_4$.
N_5^1 Abelian.
N_5^2 $[x_1, x_2] = x_3$.
N_5^3 $[x_1, x_2] = x_3$, $[x_1, x_3] = x_4$.
N_5^4 $[x_1, x_2] = x_5$, $[x_3, x_4] = x_5$.
N_5^5 $[x_1, x_2] = x_3$, $[x_1, x_3] = x_5$, $[x_2, x_4] = x_5$.
N_5^6 $[x_1, x_2] = x_3$, $[x_1, x_3] = x_4$, $[x_1, x_4] = x_5$, $[x_2, x_3] = x_5$.
N_5^7 $[x_1, x_2] = x_3$, $[x_1, x_3] = x_4$, $[x_1, x_4] = x_5$.
N_5^8 $[x_1, x_2] = x_4$, $[x_1, x_3] = x_5$.
N_5^9 $[x_1, x_2] = x_3$, $[x_1, x_3] = x_4$, $[x_2, x_3] = x_5$.

There are nine 6-dimensional nilpotent Lie algebras denoted N_6^k for $k = 1, \ldots, 9$ which are the direct sum of N_5^k and a 1-dimensional abelian ideal. Consequently, we get the following Lie algebras:-

N_6^{10} $[x_1, x_2] = x_3$, $[x_1, x_3] = x_6$, $[x_4, x_5] = x_6$.
N_6^{11} $[x_1, x_2] = x_3$, $[x_1, x_3] = x_4$, $[x_1, x_4] = x_6$, $[x_2, x_3] = x_6$, $[x_2, x_5] = x_6$.
N_6^{12} $[x_1, x_2] = x_3$, $[x_1, x_3] = x_4$, $[x_1, x_4] = x_6$, $[x_2, x_5] = x_6$.
N_6^{13} $[x_1, x_2] = x_3$, $[x_1, x_3] = x_5$, $[x_2, x_4] = x_5$, $[x_1, x_5] = x_6$, $[x_3, x_4] = x_6$.

N_6^{14} $[x_1, x_2] = x_3$, $[x_1, x_3] = x_4$, $[x_1, x_4] = x_5$, $[x_2, x_3] = x_5$, $[x_2, x_5] = x_6$, $[x_3, x_4] = -x_6$.
N_6^{15} $[x_1, x_2] = x_3$, $[x_1, x_3] = x_4$, $[x_1, x_4] = x_5$, $[x_2, x_3] = x_5$, $[x_1, x_5] = x_6$, $[x_2, x_4] = x_6$.
N_6^{16} $[x_1, x_2] = x_3$, $[x_1, x_3] = x_4$, $[x_1, x_4] = x_5$, $[x_2, x_5] = x_6$, $[x_3, x_4] = -x_6$.
N_6^{17} $[x_1, x_2] = x_3$, $[x_1, x_3] = x_4$, $[x_1, x_4] = x_5$, $[x_1, x_5] = x_6$, $[x_2, x_3] = x_6$.
N_6^{18} $[x_1, x_2] = x_3$, $[x_1, x_3] = x_4$, $[x_1, x_4] = x_5$, $[x_1, x_5] = x_6$.
$N_6^{19}(a)$ $[x_1, x_2] = x_4$, $[x_1, x_3] = x_5$, $[x_2, x_4] = x_6$, $[x_3, x_5] = ax_6$.
N_6^{20} $[x_1, x_2] = x_4$, $[x_1, x_3] = x_5$, $[x_1, x_5] = x_6$, $[x_2, x_4] = x_6$.
$N_6^{21}(a)$ $[x_1, x_2] = x_3$, $[x_1, x_3] = x_4$, $[x_2, x_3] = x_5$, $[x_1, x_4] = x_6$, $[x_2, x_5] = ax_6$.
$N_6^{22}(a)$ $[x_1, x_2] = x_5$, $[x_1, x_3] = x_6$, $[x_2, x_4] = ax_6$, $[x_3, x_4] = x_5$.
N_6^{23} $[x_1, x_2] = x_3$, $[x_1, x_3] = x_5$, $[x_1, x_4] = x_6$, $[x_2, x_4] = x_5$.
$N_6^{24}(a)$ $[x_1, x_2] = x_3$, $[x_1, x_3] = x_5$, $[x_1, x_4] = ax_6$, $[x_2, x_3] = x_6$,$[x_2, x_4] = x_5$.
N_6^{25} $[x_1, x_2] = x_3$, $[x_1, x_3] = x_5$, $[x_1, x_4] = x_6$.
N_6^{26} $[x_1, x_2] = x_4$, $[x_1, x_3] = x_5$, $[x_2, x_3] = x_6$.

Note that for all classes that depend on a parameter a, the Lie algebra with parameter a is isomorphic to the Lie algebra (from the same class) with parameter b if and only if there is an $\alpha \in F^*$ with $a = \alpha^2 b$.

106.18.4 Intrinsics for Working with the Classifications

`SolvableLieAlgebra( F, n, k :` *parameters*`)`

This function returns the solvable Lie algebra L_n^k over the field F. The multiplication table is exactly the same as that given in the classification of solvable algebras above, where the basis element x_i corresponds to the i-th basis element of the Lie algebra returned.

`pars` SeqEnum *Default :* []

If the Lie algebra L_n^k depends on one or more parameters, then the parameter `pars` specifies the parameter values corresponding to the Lie algebra which is required.

Example H106E53

```
> F<a>:= RationalFunctionField( Rationals() );
> K:= SolvableLieAlgebra( F, 3, 3 : pars:= [a] );
> K.3*K.1;
(0 1 0)
> K.3*K.2;
(a 1 0)
```

`NilpotentLieAlgebra( F, r, k : parameters)`

This function returns the nilpotent Lie algebra N_r^k over the field F. The multiplication table is exactly the same as that given in the classification of nilpotent algebras above, where the basis element x_i corresponds to the i-th basis element of the Lie algebra returned.

`pars` SeqEnum *Default :* []

If the Lie algebra N_r^k depends upon one or more parameters, then the parameter `pars` specifies the parameter values corresponding to the Lie algebra which is required.

Example H106E54

```
> F<a>:= RationalFunctionField( Rationals() );
> K:= NilpotentLieAlgebra( F, 6, 19 : pars:= [a^3] );
> K.3*K.5;
(  0   0   0   0   0 a^3)
```

`AllSolvableLieAlgebras(F, d)`

Given a finite field F and d an integer equal to 2, 3 or 4, this function returns a sequence containing all solvable Lie algebras of dimension d over the field F.

`AllNilpotentLieAlgebras(F, d)`

Given a finite field F and d an integer equal to 3, 4, 5 or 6, this function returns a sequence containing all nilpotent Lie algebras of dimension d over the field F. If the dimension is 6 then the characteristic of F may not be 2.

`IdDataSLAC(L)`

Given a solvable Lie algebra L of dimension 2, 3, or 4, this function returns data that identifies L with the isomorphic algebra in the classification of solvable Lie algebras. (SLAC stands for Solvable Lie Algebras Classification.) Three objects are returned: a string, a sequence and a map.

The string gives the name of the Lie algebra as it occurs in the classification, with information about the field and the parameters.

The sequence contains the parameters of the Lie algebra in the classification to which L is isomorphic.

The map is an isomorphism from L to the corresponding Lie algebra contained in the classification.

`IdDataNLAC(L)`

Given a nilpotent Lie algebra L of dimension 3, 4, 5 or 6 this function returns data that identifies L with the isomorphic algebra in the classification of nilpotent Lie algebras. (NLAC stands for Nilpotent Lie Algebras Classification.) Three objects are returned: a string giving the name of the algebra N in the classification, a sequence giving the parameters for N, and the isomorphism mapping L to N.

`MatrixOfIsomorphism(f)`

Given an isomorphism f as returned by either `IdDataSLAC` or `IdDataNLAC`, this function returns the matrix of that isomorphism. The row convention is used, i.e., the i-th row contains the coordinates of the image of the i-th basis element of the domain of f.

Example H106E55

We define a solvable Lie algebra of dimension 4 that depends on a parameter a. We identify this Lie algebra as a member of the classification.

```
> F<a>:= RationalFunctionField( Rationals() );
> T:= [ <1,2,2,1>, <1,2,3,a>, <1,4,4,a>, <2,1,2,-1>, <2,1,3,-a>, <4,1,4,-a> ];
> L:= LieAlgebra< F, 4 | T >;
> s,p,f:= IdDataSLAC( L );
> s;
L4_6( Univariate rational function field over Rational Field
Variables: a, 0, -a/(a^2 + 2*a + 1) )
> p;
[
    0,
    -a/(a^2 + 2*a + 1)
]
> MatrixOfIsomorphism( f );
[0   (-a - 1)/(a - 1)   (-a - 1)/(a - 1)   (a^2 + 2*a + 1)/(a^2 - a)]
[0   -1/(a - 1)   -a/(a - 1)   (a + 1)/(a - 1)]
[0   -1/(a^2 - 1)   -a/(a^2 - 1)   a/(a - 1)]
[1/(a + 1)   0   0   0]
```

So generically, the Lie algebra is isomorphic to $L_4^6(0, -a/(a^2+2a+1))$. We see that the parameters are not defined if $a = -1$. Furthermore, the isomorphism is not defined if $a = \pm 1$, or $a = 0$. We investigate those cases.

```
> a:= 1;
> T:= [ <1,2,2,1>, <1,2,3,a>, <1,4,4,a>, <2,1,2,-1>, <2,1,3,-a>, <4,1,4,-a> ];
> L:= LieAlgebra< Rationals(), 4 | T >;
> s,p,f:= IdDataSLAC( L );
> s;
L4_3( Rational Field, 0 )
> MatrixOfIsomorphism( f );
[ 0  1  1  0]
```

```
[ 0  0 -1  1]
[ 0  0  0  1]
[ 1  0  0  0]
> a:= -1;
> T:= [ <1,2,2,1>, <1,2,3,a>, <1,4,4,a>, <2,1,2,-1>, <2,1,3,-a>, <4,1,4,-a> ];
> L:= LieAlgebra< Rationals(), 4 | T >;
> s,p,f:= IdDataSLAC( L );
> s;
L4_7( Rational Field, 0, 1 )
> MatrixOfIsomorphism( f );
[   0  1/2  1/2  1/2]
[   0  1/2 -1/2 -1/2]
[   0  1/2 -1/2  1/2]
[   1    0    0    0]
> a:= 0;
> T:= [ <1,2,2,1>, <1,2,3,a>, <1,4,4,a>, <2,1,2,-1>, <2,1,3,-a>, <4,1,4,-a> ];
> L:= LieAlgebra< Rationals(), 4 | T >;
> s,p,f:= IdDataSLAC( L );
> s;
L4_4( Rational Field )
> MatrixOfIsomorphism( f );
[ 0  0  1  0]
[ 0  1  0 -1]
[ 0  1  0  0]
[ 1  0  0  0]
```

We see that for $a = 1$ the Lie algebra is isomorphic to $L_4^3(0)$, and the isomorphism is defined over any field. If $a = -1$, then the Lie algebra is isomorphic to $L_4^7(0, 1)$. However, the isomorphism is not defined if the characteristic of the field is 2. But then we are back in the case $a = 1$. Finally, for $a = 0$ the Lie algebra is isomorphic to L_4^4.

Example H106E56

The positive part of the simple Lie algebra of type G_2 is a nilpotent Lie algebra of dimension six. We identify it in the classification of nilpotent algebras, both in characteristic 0, and in characteristic 3.

```
> L:= LieAlgebra( "G2", Rationals() );
> x,y,h:= ChevalleyBasis( L );
> x;
[ (0 0 0 0 0 0 0 0 1 0 0 0 0 0), (0 0 0 0 0 0 0 0 0 0 1 0 0 0 0), (0 0 0
0 0 0 0 0 0 0 1 0 0 0), (0 0 0 0 0 0 0 0 0 0 0 1 0 0), (0 0 0 0 0 0 0
0 0 0 0 0 1 0), (0 0 0 0 0 0 0 0 0 0 0 0 0 0 1) ]
```

So we see that the positive part is spanned by basis vectors `L.i` with i=9, 10, 11, 12, 13 and 14. So

```
> K:= sub< L | [ L.i : i in [9,10,11,12,13,14] ] >;
> name,pp,f:= IdDataNLAC( K );
> name;
```

```
N6_16( Rational Field )
> MatrixOfIsomorphism( f );
[  1   0   0   0   0   0]
[  0   1   0   0   0   0]
[  0   0   1   0   0   0]
[  0   0   0 1/2   0   0]
[  0   0   0   0 1/6   0]
[  0   0   0   0   0 1/6]
> L:= LieAlgebra( "G2", GF(3) );
> K:= sub< L | [ L.i : i in [9,10,11,12,13,14] ] >;
> name,pp,f:= IdDataNLAC( K );
> name;
N6_19( Finite field of size 3, 0 )
```

We see that in characteristic 3 L is isomorphic to a Lie algebra from a different class.

106.19 Semisimple Subalgebras of Simple Lie Algebras

Here we describe the functions for working with the classification of the semisimple subalgebras of the simple Lie algebras. These subalgebras have been classified for the simple Lie algebras over the complex numbers, of ranks up to 8. They have been classified up to linear equivalence. Two subalgebras K_1, K_2 of a Lie algebra L are linearly equivalent if for every representation of L the induced representations of K_1, K_2 are equivalent. The basic function for dealing with them returns a directed graph, describing the inclusions among the subalgebras, and having the subalgebras as labels of the vertices. (We refer to [dG11] for the background details of this classification.)

`SubalgebrasInclusionGraph( t )`

Here t has to be a simple type of rank not exceeding 8. This function returns a directed graph G. The vertices of this graph are numbered from 1 to the number of semisimple subalgebras. Furthermore, the last vertex is numbered 0. A vertex has a label that is the semisimple subalgebra corresponding to it. The label of the last vertex (numbered 0), has the Lie algebra L of type t as its label. All other semisimple Lie algebras are subalgebras of this one.

The Lie algebra L (and its subalgebras) is defined over the rational numbers, or over a cyclotomic field. It is sometimes necessary to take an extension, because for some types not all subalgebras are defined over the rationals.

Moreover, in G there is an edge from the vertex with label K_1 to the vertex with label K_2 if and only if K_1 has a subalgebra that is linearly equivalent (as subalgebra of L) to K_2. We remark that it does not mean that K_2 is a subalgebra of K_1 (rather that it is linearly equivalent to a subalgebra of K_1).

Example H106E57

We consider the subalgebras of the Lie algebra of type C_3. We compute the types of its maximal subalgebras.

```
> G:= SubalgebrasInclusionGraph( "C3" );
> G;
Digraph
Vertex  Neighbours
1       ;
2       ;
3       ;
4       ;
5       ;
6       ;
7       ;
8       2 4 ;
9       5 10 ;
10      1 2 ;
11      1 2 3 ;
12      1 5 6 ;
13      3 4 6 ;
14      10 11 ;
15      9 12 14 ;
0       7 8 13 15 ;
> v:= Vertices(G);
> Label( v[10] );
Lie Algebra of dimension 6 with base ring Rational Field
> SemisimpleType( Label( v[7] ) );
A1
> SemisimpleType( Label( v[8] ) );
A2
> SemisimpleType( Label( v[13] ) );
A1 A1
> SemisimpleType( Label( v[15] ) );
A1 C2
```

`RestrictionMatrix( G, k )`

Here G is a subalgebras inclusion graph of the simple Lie algebra L, as output by the previous function, and k is a nonzero integer, corresponding to a vertex. This function returns the restriction matrix corresponding to L and the Lie algebra that is the label of the k-th vertex of G. This restriction matrix maps weights in a representation of L to weights of the subalgebra, and can be used to decompose a representation of L, as a representation of the subalgebra.

Example H106E58

We decompose the adjoint representation of the Lie algebra of type D_4, when viewed as a representation of its subalgebra of type G_2.

```
> G:= SubalgebrasInclusionGraph( "D4" );
> v:= Vertices(G);
> tt:= [ SemisimpleType( Label(a) ) : a in v ];
> Index( tt, "G2" );
17
> M:= RestrictionMatrix( G, 17 );
> R:= RootDatum( "D4" : Isogeny:= "SC" );
> S:= RootDatum( "G2" : Isogeny:= "SC" );
> D:= AdjointRepresentationDecomposition(R);
> E:= Branch( S, D, M );
> WeightsAndMultiplicities(E);
[
    (0 1),
    (1 0)
]
[ 1, 2 ]
```

106.20 Nilpotent Orbits in Simple Lie Algebras

Take a simple Lie algebra over the complex numbers, and consider the connected component of its automorphism group that contains the identity. This group acts on the Lie algebra, and there is interest in understanding the nature of its orbits. The nilpotent orbits for simple Lie algebras have been classified. We refer to the book by Collingwood and McGovern [CM93] for the details of this classification.

The main technical tools used for the classification are the weighted Dynkin diagram and the sl_2-triple. The weighted Dynkin diagram is the Dynkin diagram of the root system of the Lie algebra, with labels that can be 0, 1, 2. A nilpotent orbit is uniquely determined by its weighted Dynkin diagram. By the Jacobson-Morozov theorem a nilpotent element of a semisimple Lie algebra can be embedded (as nilpositive element) in an sl_2-triple. Now two nilpotent elements are conjugate (under the group) if and only if the corresponding sl_2-triples are conjugate. This yields a bijection between nilpotent orbits and conjugacy classes of simple subalgebras isomorphic to sl_2.

This section describes functions for working with the classification of nilpotent orbits in simple Lie algebras. One of the main invariants of a nilpotent orbit is its weighted Dynkin diagram. We represent such a diagram by a sequence of its labels; they are mapped to the nodes of the Dynkin diagram in the order determined by the Cartan matrix of the root datum. Also, an sl_2-triple is represented by a sequence $[f, h, e]$ of three elements of a Lie algebra; these satisfy the commutation relations $[h, e] = 2e$, $[h, f] = -2f$, $[e, f] = h$.

Throughout this section we consider orbits that are not the zero orbit.

`IsGenuineWeightedDynkinDiagram( L, wd )`

Given a simple Lie algebra L, and a sequence wd consisting of integers that are 0, 1, or 2, this function returns `true` if wd corresponds to a nilpotent orbit (in other words, if it is the weighted Dynkin diagram of a nilpotent orbit). If wd does corresponds to a nilpotent orbit, an sl_2-triple in L, such that the third element lies in the nilpotent orbit corresponding to the weighted Dynkin diagram is returned. If wd does not correspond to a nilpotent orbit. the second return value is a sequence consisting of three zeros of L.

Example H106E59

We can use this function to find the classification of the nilpotent orbits of a given Lie algebra. First we construct all possible weighted Dynkin diagrams, and then we remove those that do not correspond to an orbit.

```
> L:= LieAlgebra( RootDatum("D4"), Rationals() );
> [ w : i,j,k,l in [0,1,2] | IsGenuineWeightedDynkinDiagram(L, w)
>     where w := [i,j,k,l] ];
[
    [ 0, 0, 0, 2 ],
    [ 0, 0, 2, 0 ],
    [ 0, 1, 0, 0 ],
    [ 0, 2, 0, 0 ],
    [ 0, 2, 0, 2 ],
    [ 0, 2, 2, 0 ],
    [ 1, 0, 1, 1 ],
    [ 2, 0, 0, 0 ],
    [ 2, 0, 2, 2 ],
    [ 2, 2, 0, 0 ],
    [ 2, 2, 2, 2 ]
]
```

`NilpotentOrbit( L, wd )`

This returns the nilpotent orbit in the simple Lie algebra L with weighted Dynkin diagram given by the sequence wd. It is *not* checked whether the weighted Dynkin diagram really corresponds to a nilpotent orbit.

`NilpotentOrbit( L, e )`

This returns the nilpotent orbit in the simple Lie algebra L having representative e. Here e has to be a nilpotent element of the Lie algebra. This condition is not checked by the function.

Example H106E60

```
> L:= LieAlgebra( RootDatum("A2"), Rationals() );
> NilpotentOrbit( L, [2,2] );
Nilpotent orbit in Lie algebra of type A2
> NilpotentOrbit( L, L.1 );
Nilpotent orbit in Lie algebra of type A2
```

`NilpotentOrbits( L )`

Given a simple Lie algebra L, this function returns the sequence of all nilpotent orbits in the simple Lie algebra L.

Example H106E61

We compute the nilpotent orbits of the Lie algebra of type D_4, and observe that they are the same as those found in Example H106E59.

```
> L:= LieAlgebra( RootDatum("D4"), Rationals() );
> o:= NilpotentOrbits(L);
> [ WeightedDynkinDiagram(orb) : orb in o ];
[
    [ 2, 2, 2, 2 ],
    [ 2, 0, 2, 2 ],
    [ 2, 2, 0, 0 ],
    [ 0, 2, 0, 2 ],
    [ 0, 2, 2, 0 ],
    [ 0, 2, 0, 0 ],
    [ 1, 0, 1, 1 ],
    [ 2, 0, 0, 0 ],
    [ 0, 0, 0, 2 ],
    [ 0, 0, 2, 0 ],
    [ 0, 1, 0, 0 ]
]
```

`Partition( o )`

Here o is a nilpotent orbit in a simple Lie algebra of classical type (i.e., of type A_n, B_n, C_n or D_n). The nilpotent orbits for the Lie algebras of these types have been classified in terms of partitions. This function returns the partition corresponding to the orbit.

Example H106E62

```
> L:= LieAlgebra( RootDatum("D4"), Rationals() );
> orbs:= NilpotentOrbits( L );
> Partition( orbs[5] );
[ 4, 4 ]
> Partition( orbs[6] );
[ 3, 3, 1, 1 ]
```

`SL2Triple( o )`

Given a nilpotent orbit o in a simple Lie algebra L, this function returns an sl_2-triple, $[f, h, e]$ of elements of L, such that e lies in the nilpotent orbit.

`SL2Triple( L, e )`

Given a semisimple Lie algebra L of characteristic 0, and a nilpotent element e of L, this function returns an sl_2-triple $[f, h, e]$ of elements of L. It may also work for other Lie algebras, and in other characteristics, but this is not guaranteed.

`Representative( o )`

Given a nilpotent orbit o for a simple Lie algebra L, this function returns an $e \in L$ lying in the orbit.

`WeightedDynkinDiagram( o )`

Given a nilpotent orbit o for a simple Lie algebra L, this function returns its weighted Dynkin diagram.

Example H106E63

We take some nilpotent element in the Lie algebra of type E_8 and we find the weighted Dynkin diagram of the orbit it lies in.

```
> L:= LieAlgebra( RootDatum("E8"), Rationals() );
> x,_,_:= ChevalleyBasis(L);
> orb:= NilpotentOrbit( L, x[1]+x[10]-x[30]+3*x[50]-2*x[100] );
> WeightedDynkinDiagram( orb );
[ 1, 0, 0, 0, 0, 0, 0, 1 ]
```

106.21 Bibliography

[**CM93**] David H. Collingwood and William M. McGovern. *Nilpotent orbits in semisimple Lie algebras*. Van Nostrand Reinhold Mathematics Series. Van Nostrand Reinhold Co., New York, 1993.

[**CM09**] Arjeh M. Cohen and Scott H. Murray. An algorithm for Lang's Theorem. *Journal of Algebra*, 322:675–702, 2009.

[**CR09**] Arjeh M. Cohen and Dan Roozemond. Computing Chevalley bases in small characteristics. *J. Algebra*, 322(3):703–721, August 2009.

[**CSUW01**] Arjeh M. Cohen, Anja Steinbach, Rosane Ushirobira, and David Wales. Lie algebras generated by extremal elements. *J. Algebra*, 236(1):122–154, 2001.

[**dG00**] W.A. de Graaf. *Lie Algebras: Theory and Algorithms*. Number 56 in North-Holland Mathematical Library. Elsevier, 2000.

[**dG05**] W. A. de Graaf. Classification of solvable Lie algebras. *Experimental Mathematics*, 14(1):15–25, 2005.

[**dG07**] W. A. de Graaf. Classification of 6-dimensional nilpotent Lie algebras over fields of characteristic not 2. *Journal of Algebra*, 309(2):640–653, 2007.

[**dG11**] Willem A. de Graaf. Constructing semisimple subalgebras of semisimple Lie algebras. *J. Algebra*, 325:416–430, 2011.

[**Hog82**] G. M. D. Hogeweij. Almost-classical Lie algebras. I, II. *Nederl. Akad. Wetensch. Indag. Math.*, 44(4):441–452, 453–460, 1982.

[**Jac62**] N. Jacobson. *Lie algebras*. Interscience Tracts in Pure and Applied Mathematics, No. 10. Interscience Publishers New York-London, 1962.

[**Kos66**] B. Kostant. Groups over Z. In *Algebraic Groups and Discontinuous Subgroups (Proc. Sympos. Pure Math., Boulder, Colo., 1965)*, pages 90–98. Amer. Math. Soc., Providence, R.I., 1966.

[**Roo10**] D.A. Roozemond. *Algorithms for Lie algebras of algebraic groups*. PhD thesis, Technische Universiteit Eindhoven, 2010.

[**Roo11**] Dan Roozemond. On Lie algebras generated by few extremal elements. *J. Algebra*, 348:462–476, 2011.

[**SF88**] Helmut Strade and Rolf Farnsteiner. *Modular Lie algebras and their representations*, volume 116 of *Monographs and Textbooks in Pure and Applied Mathematics*. Marcel Dekker Inc., New York, 1988.

[**Str04**] Helmut Strade. *Simple Lie algebras over fields of positive characteristic. I*, volume 38 of *de Gruyter Expositions in Mathematics*. Walter de Gruyter & Co., Berlin, 2004. Structure theory.

[**ZK90**] E.I. Zel′manov and A.I. Kostrikin. A theorem on sandwich algebras. *Trudy Mat. Inst. Steklov.*, 183:106–111, 225, 1990. Translated in Proc. Steklov Inst. Math. **1991**, no. 4, 121–126, Galois theory, rings, algebraic groups and their applications (Russian).

107 KAC-MOODY LIE ALGEBRAS

107.1 Introduction 3299

107.2 Generalized Cartan Matrices 3300

`IsGeneralizedCartanMatrix(C)` 3300
`KacMoodyClass(C)` 3300
`KacMoodyClasses(C)` 3300

107.3 Affine Kac-Moody Lie Algebras 3301

107.3.1 Constructing Affine Kac-Moody Lie Algebras 3301

`AffineLieAlgebra(N, F)` 3301
`AffineLieAlgebra(C, F)` 3301

107.3.2 Properties of Affine Kac-Moody Lie Algebras 3302

`CartanMatrix(L)` 3302
`CartanName(L)` 3302
`Dimension(L)` 3302
`CoefficientRing(L)` 3302
`FiniteLieAlgebra(L)` 3302
`LaurentSeriesRing(L)` 3302
`StandardGenerators(L)` 3302

107.3.3 Constructing Elements of Affine Kac-Moody Lie Algebras 3303

`.` 3303
`HasAttribute(L, "c")` 3303
`HasAttribute(L, "d")` 3303
`elt< >` 3303

107.3.4 Properties of Elements of Affine Kac-Moody Lie Algebras 3304

`EltTup(x)` 3304
`IsZero(x)` 3304
`eq` 3304
`+` 3304
`-` 3304
`*` 3304
`-x` 3304

107.4 Bibliography 3305

Chapter 107
KAC-MOODY LIE ALGEBRAS

107.1 Introduction

Lie algebras of finite dimension are well understood, and numerous procedures for performing calculations with them are described in Chapter 106. An important class of infinite dimensional Lie algebras is that of *Kac-Moody* Lie algebras. The principal text on this subject is a book by Kac [Kac90]. Let us briefly introduce these Lie algebras.

A *generalized Cartan matrix* is an integral matrix $A = (a_{ij})_{i,j=1}^{n}$ such that $a_{ii} = 2$, $a_{ij} < 0$ for $i \neq j$, and $a_{ij} = 0$ implies $a_{ji} = 0$. (Note that in particular, a Cartan matrix in the usual sense is a generalized Cartan matrix.)

To a generalized Cartan matrix we associate a Kac-Moody Lie algebra $\mathfrak{g}(A)$. This Lie algebra is generated by $3n$ elements e_i, f_i, h_i $(i = 1, \ldots, n)$ satisfying the following defining relations:

$$[h_i, h_j] = 0, [e_i, f_i] = h_i, [e_i, f_j] = 0 \text{ if } i \neq j,$$

$$[h_i, e_j] = a_{ij} e_j, [h_i, f_j] = -a_{ij} f_j,$$

$$(\mathrm{ad} e_i)^{1-a_{ij}} e_j = 0, (\mathrm{ad} f_i)^{1-a_{ij}} f_j = 0 \text{ if } i \neq j.$$

The class of Kac-Moody Lie algebras breaks up into three subclasses:

(a) There is a vector θ of positive integers such $A\theta$ is a positive vector. In this case the Lie algebra $\mathfrak{g}(A)$ is finite-dimensional and reductive.

(b) There is a vector δ of positive integers such that $A\delta = 0$. In this case $\mathfrak{g}(A)$ is infinite-dimensional, but is of polynomial growth. These Lie algebras are called *affine Lie algebras.*

(c) There is a vector α of positive integers such that $A\alpha$ is negative. In this case $\mathfrak{g}(A)$ is infinite-dimensional and of exponential growth.

The procedures for finite-dimensional Lie algebras are described in Chapter 106. The affine Lie algebras are described in Section 107.3. The Kac-Moody Lie algebras of type (c) are not yet available.

107.2 Generalized Cartan Matrices

IsGeneralizedCartanMatrix(C)

Whether the square matrix C is a generalized Cartan matrix.

KacMoodyClass(C)

The class of the indecomposable generalized Cartan matrix C. The first return value is a string, "a", "b" or "c", corresponding to the three cases described in the introduction 107.1. The second is a positive integral column vector v such that Cv is positive, 0 or negative, respectively (so this return value corresponds to the vectors θ, δ and α in the introduction).

KacMoodyClasses(C)

The class of the possibly decomposable generalized Cartan matrix C. Three sequences are returned: the first is a sequence of strings "a", "b" or "c", describing the class of each component; the second is a positive integral vector v such that Cv is positive, 0 or negative, respectively (see KacMoodyClass).

The third sequence Q contains integral sequences Q_i such that the i-th component is formed by taking the rows and columns with index j, for $j \in Q_i$.

Example H107E1

First, we consider an indecomposable Cartan matrix.

```
> C := Matrix(Integers(), 3, 3, [2,-1,0, -5,2,-1, 0,-1,2]);
> s, v := KacMoodyClass(C);
> s;
c
> v;
[2]
[5]
[1]
> C*v;
[-1]
[-1]
[-3]
```

As a second example, we consider a decomposable Cartan matrix.

```
> C := CartanMatrix("B2 A~3");
> S, V, Q := KacMoodyClasses(C);
> S;
[ a, b ]
> Q;
[
    [ 1, 2 ],
    [ 3, 4, 5, 6 ]
]
```

```
> C1 := Submatrix(C, Q[1], Q[1]);
> KacMoodyClass(C1);
a
> C2 := Submatrix(C, Q[2], Q[2]);
> KacMoodyClass(C2);
b
```

107.3 Affine Kac-Moody Lie Algebras

For affine Lie algebras there exists a well-known explicit construction of these in terms of an underlying finite-dimensional Lie algebra and a central extension (see [Kac90, Chapters 7,8]). We briefly reiterate the construction here. Suppose A is an affine Cartan matrix, so that $\mathfrak{g}(A)$ is an affine Lie algebra; then A is of affine Cartan type $\tilde{\mathrm{X}}_n$ for X=A,B,C,D,E,F, or G, and some n. If we let $\mathfrak{g}_0$ be the finite variant (i.e., a Lie algebra of Cartan type X_n) then

$$\mathfrak{g}(A) \cong \mathfrak{g}_0 \otimes \mathbf{C}[t, t^{-1}] \oplus \mathbf{C}c \oplus \mathbf{C}d$$

for some formal basis elements c and d, where $\mathbf{C}[t, t^{-1}]$ is the ring of Laurent polynomials over $\mathbf{C}$. In MAGMA we represent affine Lie algebras and their elements using the form on the right hand side.

Multiplication is given by

$$[t^k \otimes x \oplus \lambda c \oplus \mu d, t^{k_1} \otimes y \oplus \lambda_1 c \oplus \mu_1 d] =$$

$$(t^{k+k_1} \otimes [x, y] + \mu k_1 t^{k_1} \otimes y - \mu_1 k t^k \otimes x) \oplus k\delta_{k,-k_1}(x|y)c,$$

where $(x|y)$ denotes a fixed non-degenerate invariant symmetric bilinear $\mathbf{C}$-valued form on $\mathfrak{g}_0$.

If we fix E_i, F_i to be canonical generators of g_0, then the canonical generators of $\mathfrak{g}(A)$, as described in the introduction (107.1) are given by $e_0 = t \otimes E_0$, $f_0 = t^{-1} \otimes F_0$, and $e_i = 1 \otimes E_i$, $f_i = 1 \otimes F_i$, for $i = 1, \ldots, l$, where l is the rank of the Cartan matrix.

Affine Lie algebras and their elements are of type `AlgKac` and `AlgKacElt` respectively.

107.3.1 Constructing Affine Kac-Moody Lie Algebras

`AffineLieAlgebra(N, F)`

Construct the affine Kac-Moody Lie algebra of type N over the field F. N should be a string describing an affine Cartan type (e.g. `A~3`). See Section 101.6 for more information on the conventions, syntax, and functions for creating and working with affine Cartan matrices.

`AffineLieAlgebra(C, F)`

Construct the affine Kac-Moody Lie algebra with affine Cartan matrix C over the field F.

Example H107E2

We demonstrate the construction functions.

```
> L := AffineLieAlgebra("G~2", Rationals());
> L;
Affine Kac-Moody Lie algebra over Rational Field
> C := Matrix(Integers(),3,3,[2,-1,-1,-1,2,-1,-1,-1,2]);
> CartanName(C);
A~2
> L := AffineLieAlgebra(C, Rationals());
> L;
Affine Kac-Moody Lie algebra over Rational Field
```

107.3.2 Properties of Affine Kac-Moody Lie Algebras

`CartanMatrix(L)`

The Cartan matrix of L.

`CartanName(L)`

The Cartan type of L.

`Dimension(L)`

Infinity.

`CoefficientRing(L)`

The coefficient ring of L.

`FiniteLieAlgebra(L)`

The Lie algebra $\mathfrak{g}_0$ underlying L (see the Introduction, Section 107.1).

`LaurentSeriesRing(L)`

The Laurent series ring $\mathbf{C}[t, t^{-1}]$ underlying L (see the Introduction, Section 107.1).

`StandardGenerators(L)`

The standard generators of L. These are returned as three sequences, the first containing the e_i, the second containing the f_i, and the last containing the h_i. Note that the root usually labeled "0" occurs as the last element of each of these sequences.

Example H107E3

We demonstrate some properties of affine Lie algebras.

```
> L := AffineLieAlgebra("A~2", Rationals());
> L;
Affine Kac-Moody Lie algebra over Rational Field
> Lf := FiniteLieAlgebra(L);
> Lf;
Lie Algebra of dimension 8 with base ring Rational Field
> SemisimpleType(Lf);
A2
> e,f,h := StandardGenerators(L);
> e;
[ (0 0 0 0 0 1 0 0), (0 0 0 0 0 0 1 0), (t)*(1 0 0 0 0 0 0 0) ]
> F<e1,e2,e0,f1,f2,f0> := FreeLieAlgebra(Rationals(), 6);
> phi := hom<F -> L | e cat f>;
> phi(e1);
(0 0 0 0 0 1 0 0)
> phi(e1*e0) eq phi(e1)*phi(e0);
true
```

107.3.3 Constructing Elements of Affine Kac-Moody Lie Algebras

`L . i`

The i-th basis element of the finite dimensional Lie algebra underlying L, as an element of L.

`HasAttribute(L, "c")`

`HasAttribute(L, "d")`

Return `true` and the basis element c or d of L, according to the second argument of `HasAttribute`.

`elt< L | <[<`p_1, y_1`>, ...], `λ, μ`> >`

For a 3-tuple t such that t_1 is a sequence of elements of $\mathbf{C}[t, t^{-1}] \times \mathfrak{g}_0$, and t_2 and t_3 are elements of the coefficient ring of L, construct

$$\sum_{(p,y)\in t_1} p \otimes y \oplus t_2 c \oplus t_3 d \in L.$$

See `EltTup` below for the converse function.

107.3.4 Properties of Elements of Affine Kac-Moody Lie Algebras

`EltTup(x)`

The element x of the affine Lie algebra L as a three-tuple t such that

$$x = \sum_{(p,y)\in t_1} p \otimes y \oplus t_2 c \oplus t_3 d.$$

The first entry, t_1, is a sequence of pairs $(p, y) \in \mathbf{C}[t, t^{-1}] \times \mathfrak{g}_0$, t_2 is the coefficient of c and t_3 is the coefficient of d.

`IsZero(x)`

Whether x is zero.

`x eq y`

Whether x and y are equal.

`x + y`
`x - y`
`x * y`

Respectively the sum, difference, and multiplication of x and y.

`-x`

The negation of x.

Example H107E4

We perform various computations with elements of an affine Lie algebra.

```
> L<t> := AffineLieAlgebra("B~3", Rationals());
> Lf := FiniteLieAlgebra(L);
> e,f,h := StandardGenerators(L);
> E,F,H := StandardBasis(Lf);
> e[1] eq L!E[1];
true
> x := e[4];
> x;
(t)*(1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0)
> EltTup(x);
<[
    <t, (1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0)>
], 0, 0>
> elt<L | EltTup(x) > eq x;
true
> y := elt<L | <[<t^2-t^-2, F[1]>,<-2,Lf.3>], -1/3, 1> >;
> y;
```

```
(-2)*(0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0) + (-t^-2 + t^2)*(0 0
0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0) -1/3*c + d
> z := t^3*L.2 - 1/5*h[1] + 1/7*L‘c-L‘d;
> z;
(t^3)*(0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0) + (2/5)*(0 0 0 0 0 0
0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0) (-1/5)*(0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0
0 0 0 0 0) + 1/7*c -1*d
> x*(y*z) + y*(z*x) + z*(x*y);
0
```

107.4 Bibliography

[**Kac90**] Victor G. Kac. *Infinite Dimensional Lie Algebras.* Cambridge University Press, 1990.

108 QUANTUM GROUPS

108.1 Introduction 3309

108.2 Background 3309

108.2.1 Gaussian Binomials 3309

108.2.2 Quantized Enveloping Algebras . 3310

108.2.3 Representations of $U_q(L)$ 3311

108.2.4 PBW-type Bases 3311

108.2.5 The **Z***-form of $U_q(L)$ 3312*

108.2.6 The Canonical Basis 3313

108.2.7 The Path Model 3314

108.3 Gauss Numbers 3315

GaussNumber(n, v) 3315
GaussianFactorial(n, v) 3315
GaussianBinomial(n, k, v) 3315

108.4 Construction 3316

QuantizedUEA(R) 3316
QuantizedUEAlgebra(R) 3316
QuantizedUniversalEnveloping Algebra(R) 3316
AssignNames(U, S) 3317
ChangeRing(U, R) 3317

108.5 Related Structures 3317

CoefficientRing(U) 3317
RootDatum(U) 3317
PositiveRootsPerm(U) 3317

108.6 Operations on Elements . . 3318

+ - * * * ^ 3318
! 3318
Zero(U) 3318
! 3318
One(U) 3318
. 3318
! 3318
KBinomial(U, i, s) 3318
KBinomial(K, s) 3318
Monomials(u) 3318
Coefficients(u) 3319
^ 3319
Degree(u, i) 3319
KDegree(m, i) 3319

108.7 Representations 3320

HighestWeightRepresentation(U, w) 3320
HighestWeightModule(U, w) 3320
WeightsAndVectors(V) 3321
HighestWeightsAndVectors(V) 3321
CanonicalBasis(V) 3321
TensorProduct(Q) 3322

108.8 Hopf Algebra Structure . . 3323

UseTwistedHopfStructure(U, f, g) 3323
HasTwistedHopfStructure(U) 3323
Counit(U) 3323
Antipode(U) 3323
Comultiplication(U, d) 3323

108.9 Automorphisms 3324

BarAutomorphism(U) 3324
AutomorphismOmega(U) 3324
AntiAutomorphismTau(U) 3324
AutomorphismTalpha(U, k) 3324
DiagramAutomorphism(U, p) 3325
GraphAutomorphism(U, p) 3325

108.10 Kashiwara Operators . . . 3326

Falpha(m, i) 3326
Ealpha(m, i) 3326

108.11 The Path Model 3326

DominantLSPath(R, hw) 3326
Falpha(p, i) 3327
Ealpha(p, i) 3327
WeightSequence(p) 3327
RationalSequence(p) 3327
EndpointWeight(p) 3327
Shape(p) 3327
WeylWord(p) 3327
IsZero(p) 3327
eq 3327
CrystalGraph(R, hw) 3328

108.12 Elements of the Canonical Basis 3329

CanonicalElements(U, w) 3329

108.13 Homomorphisms to the Universal Enveloping Algebra . 3331

QUAToIntegralUEAMap(U) 3331

108.14 Bibliography 3332

Chapter 108
QUANTUM GROUPS

108.1 Introduction

This chapter describes the functionality for quantum groups in MAGMA. First there are a few sections that briefly describe the theoretical background behind quantum groups (or, more precisely, quantized enveloping algebras). This fixes the notation and the terminology that we use (these vary somewhat in the literature). For this we mainly follow [Jan96]. In the remainder we describe the functions that exist in MAGMA for constructing and working with quantum groups and their representations.

In MAGMA, quantized enveloping algebras have type `AlgQUE` and their elements have type `AlgQUEElt`. These types inherit from `AlgPBW` and `AlgPBWElt` respectively, which are general types for algebras with a PBW basis and their elements and inherit from `GenMPolB`, `Alg` and `Rng` and their element types.

108.2 Background

108.2.1 Gaussian Binomials

Let v be an indeterminate over $\mathbf{Q}$. For a positive integer n we set

$$[n]_v = v^{n-1} + v^{n-3} + \cdots + v^{-n+3} + v^{-n+1}.$$

We say that $[n]_v$ is the *Gaussian integer* corresponding to n. The *Gaussian factorial* $[n]_v!$ is defined by

$$[0]_v! = 1,\ [n]_v! = [n]_v[n-1]_v \cdots [1]_v \text{ for } n > 0.$$

Finally, the *Gaussian binomial* is

$$\binom{n}{k}_v = \frac{[n]!}{[k]![n-k]!}.$$

108.2.2 Quantized Enveloping Algebras

Let L be a semisimple Lie algebra with root system Φ. By $\Delta = \{\alpha_1, \ldots, \alpha_l\}$ we denote a fixed set of simple roots of Φ. Let $C = (C_{ij})$ be the Cartan matrix of Φ (with respect to Δ, i.e., $C_{ij} = \langle \alpha_i, \alpha_j^\vee \rangle$). Let $d_1, \ldots, d_l$ be the unique sequence of positive integers with greatest common divisor 1, such that $d_i C_{ji} = d_j C_{ij}$, and set $(\alpha_i, \alpha_j) = d_j C_{ij}$. (We note that this implies that (α_i, α_i) is divisible by 2.) By P we denote the weight lattice, and we extend the form $(\ ,\ )$ to P by bilinearity.

By $W(\Phi)$ we denote the Weyl group of Φ. It is generated by the simple reflections $s_i = s_{\alpha_i}$ for $1 \leq i \leq l$ (where s_α is defined by $s_\alpha(\beta) = \beta - \langle \beta, \alpha^\vee \rangle \alpha$).

We work over the field $\mathbf{Q}(q)$. For $\alpha \in \Phi$ we set

$$q_\alpha = q^{\frac{(\alpha,\alpha)}{2}},$$

and for a non-negative integer n, $[n]_\alpha = [n]_{v=q_\alpha}$; $[n]_\alpha!$ and $\binom{n}{k}_\alpha$ are defined analogously.

The quantized enveloping algebra $U_q(L)$ is the associative algebra (with one) over $\mathbf{Q}(q)$ generated by F_α, K_α, K_α^{-1}, E_α for $\alpha \in \Delta$, subject to the following relations

$$\begin{aligned} K_\alpha K_\alpha^{-1} = K_\alpha^{-1} K_\alpha = 1,\ K_\alpha K_\beta &= K_\beta K_\alpha \\ E_\beta K_\alpha &= q^{-(\alpha,\beta)} K_\alpha E_\beta \\ K_\alpha F_\beta &= q^{-(\alpha,\beta)} F_\beta K_\alpha \\ E_\alpha F_\beta &= F_\beta E_\alpha + \delta_{\alpha,\beta} \frac{K_\alpha - K_\alpha^{-1}}{q_\alpha - q_\alpha^{-1}} \end{aligned}$$

together with, for $\alpha \neq \beta \in \Delta$,

$$\sum_{k=0}^{1-\langle \beta, \alpha^\vee \rangle} (-1)^k \binom{1 - \langle \beta, \alpha^\vee \rangle}{k}_\alpha E_\alpha^{1-\langle \beta, \alpha^\vee \rangle - k} E_\beta E_\alpha^k = 0$$

$$\sum_{k=0}^{1-\langle \beta, \alpha^\vee \rangle} (-1)^k \binom{1 - \langle \beta, \alpha^\vee \rangle}{k}_\alpha F_\alpha^{1-\langle \beta, \alpha^\vee \rangle - k} F_\beta F_\alpha^k = 0.$$

The quantized enveloping algebra has an automorphism ω defined by $\omega(F_\alpha) = E_\alpha$, $\omega(E_\alpha) = F_\alpha$ and $\omega(K_\alpha) = K_\alpha^{-1}$. Also there is an anti-automorphism τ defined by $\tau(F_\alpha) = F_\alpha$, $\tau(E_\alpha) = E_\alpha$ and $\tau(K_\alpha) = K_\alpha^{-1}$. We have $\omega^2 = 1$ and $\tau^2 = 1$.

If the Dynkin diagram of Φ admits a diagram automorphism π, then π induces an automorphism of $U_q(L)$ in the obvious way (π is a permutation of the simple roots; we permute the F_α, E_α, $K_\alpha^{\pm 1}$ accordingly).

Now we view $U_q(L)$ as an algebra over $\mathbf{Q}$, and we let $\overline{\ \ } : U_q(L) \to U_q(L)$ be the automorphism defined by $\overline{F_\alpha} = F_\alpha$, $\overline{K_\alpha} = K_\alpha^{-1}$, $\overline{E_\alpha} = E_\alpha$, $\overline{q} = q^{-1}$. This map is called the *bar-automorphism.*

108.2.3 Representations of $U_q(L)$

Let $\lambda \in P$ be a dominant weight. Then there is a unique irreducible highest-weight module over $U_q(L)$ with highest weight λ. We denote it by $V(\lambda)$. It has the same character as the irreducible highest-weight module over L with highest weight λ. Furthermore, every finite-dimensional $U_q(L)$-module is a direct sum of irreducible highest-weight modules. In [Gra04] a few algorithms for constructing $V(\lambda)$ are given. In the MAGMA implementation the algorithm based on Gröbner bases is used.

It is well-known that $U_q(L)$ is a Hopf algebra. The comultiplication $\Delta : U_q(L) \to U_q(L) \otimes U_q(L)$ is defined by

$$\begin{aligned}\Delta(E_\alpha) &= E_\alpha \otimes 1 + K_\alpha \otimes E_\alpha \\ \Delta(F_\alpha) &= F_\alpha \otimes K_\alpha^{-1} + 1 \otimes F_\alpha \\ \Delta(K_\alpha) &= K_\alpha \otimes K_\alpha.\end{aligned}$$

(Note that we use the same symbol (Δ) to denote a set of simple roots of Φ; of course this does not cause confusion.) The counit $\varepsilon : U_q(L) \to \mathbf{Q}(q)$ is a homomorphism defined by $\varepsilon(E_\alpha) = \varepsilon(F_\alpha) = 0$, $\varepsilon(K_\alpha) = 1$. Finally, the antipode $S : U_q(L) \to U_q(L)$ is an anti-automorphism given by $S(E_\alpha) = -K_\alpha^{-1}E_\alpha$, $S(F_\alpha) = -F_\alpha K_\alpha$, $S(K_\alpha) = K_\alpha^{-1}$.

Using Δ we can make the tensor product $V \otimes W$ of two $U_q(L)$-modules V, W into a $U_q(L)$-module. The counit ε yields a trivial 1-dimensional $U_q(L)$-module. And with S we can define a $U_q(L)$-module structure on the dual V^* of a $U_q(L)$-module V, by $(u \cdot f)(v) = f(S(u) \cdot v)$.

The Hopf algebra structure given above is not the only one possible. For example, we can twist Δ, ε, S by an automorphism, or an anti-automorphism f. The twisted comultiplication is given by

$$\Delta^f = f \otimes f \circ \Delta \circ f^{-1},$$

the twisted antipode by

$$S^f = f \circ S \circ f^{-1},$$

if f is an automorphism, and

$$S^f = f \circ S^{-1} \circ f^{-1},$$

if f is an anti-automorphism. The twisted counit is given by $\varepsilon^f = \varepsilon \circ f^{-1}$.

108.2.4 PBW-type Bases

The first problem one has to deal with when working with $U_q(L)$ is finding a basis of it, along with an algorithm for expressing the product of two basis elements as a linear combination of basis elements. First of all we have that $U_q(L) \cong U^- \otimes U^0 \otimes U^+$ (as vector spaces), where U^- is the subalgebra generated by the F_α, U^0 is the subalgebra generated by the K_α, and U^+ is generated by the E_α. So a basis of $U_q(L)$ is formed by all elements FKE, where F, K, E run through bases of U^-, U^0, U^+ respectively.

Finding a basis of U^0 is easy: it is spanned by all $K_{\alpha_1}^{r_1} \cdots K_{\alpha_l}^{r_l}$, where $r_i \in \mathbf{Z}$. For U^- and U^+ we use the so-called *PBW-type* bases. They are defined as follows. For $\alpha, \beta \in \Delta$

we set $r_{\beta,\alpha} = -\langle \beta, \alpha^\vee \rangle$. Then for $\alpha \in \Delta$ we have the automorphism $T_\alpha : U_q(L) \to U_q(L)$ defined by

$$\begin{aligned} T_\alpha(E_\alpha) &= -F_\alpha K_\alpha \\ T_\alpha(E_\beta) &= \sum_{i=0}^{r_{\beta,\alpha}} (-1)^i q_\alpha^{-i} E_\alpha^{(r_{\beta,\alpha}-i)} E_\beta E_\alpha^{(i)}, \ (\alpha \neq \beta) \\ T_\alpha(K_\beta) &= K_\beta K_\alpha^{r_{\beta,\alpha}} \\ T_\alpha(F_\alpha) &= -K_\alpha^{-1} E_\alpha \\ T_\alpha(F_\beta) &= \sum_{i=0}^{r_{\beta,\alpha}} (-1)^i q_\alpha^{i} F_\alpha^{(i)} F_\beta F_\alpha^{(r_{\beta,\alpha}-i)}, \ (\alpha \neq \beta) \end{aligned}$$

(where $E_\alpha^{(k)} = E_\alpha^k/[k]_\alpha!$, and likewise for $F_\alpha^{(k)}$).

Let $w_0 = s_{i_1} \cdots s_{i_t}$ be a reduced expression for the longest element in the Weyl group $W(\Phi)$. For $1 \leq k \leq t$ set $F_k = T_{\alpha_{i_1}} \cdots T_{\alpha_{i_{k-1}}}(F_{\alpha_{i_k}})$, and $E_k = T_{\alpha_{i_1}} \cdots T_{\alpha_{i_{k-1}}}(E_{\alpha_{i_k}})$. Then $F_k \in U^-$, and $E_k \in U^+$. Furthermore, the elements $F_1^{m_1} \cdots F_t^{m_t}$, $E_1^{n_1} \cdots E_t^{n_t}$ (where the m_i, n_i are non-negative integers) form bases of U^- and U^+ respectively.

The elements F_α and E_α are said to have weight $-\alpha$ and α respectively, where α is a simple root. Furthermore, the weight of a product ab is the sum of the weights of a and b. Now elements of U^-, U^+ that are linear combinations of elements of the same weight are said to be homogeneous. It can be shown that the elements F_k, and E_k are homogeneous of weight $-\beta$ and β respectively, where $\beta = s_{i_1} \cdots s_{i_{k-1}}(\alpha_{i_k})$.

In the following we use the notation $F_k^{(m)} = F_k^m/[m]_{\alpha_{i_k}}!$, and $E_k^{(n)} = E_k^n/[n]_{\alpha_{i_k}}!$.

We refer to [Gra01] for an account of algorithms for expressing the product of two elements of a PBW-type basis as a linear combination of such elements. These algorithms are implemented in Magma.

108.2.5 The Z-form of $U_q(L)$

For $\alpha \in \Delta$ set

$$\binom{K_\alpha}{n} = \prod_{i=1}^{n} \frac{q_\alpha^{-i+1} K_\alpha - q_\alpha^{i-1} K_\alpha^{-1}}{q_\alpha^i - q_\alpha^{-i}}.$$

Then according to [Lus90], Theorem 6.7 the elements

$$F_1^{(k_1)} \cdots F_t^{(k_t)} K_{\alpha_1}^{\delta_1} \binom{K_{\alpha_1}}{m_1} \cdots K_{\alpha_l}^{\delta_l} \binom{K_{\alpha_l}}{m_l} E_1^{(n_1)} \cdots E_t^{(n_t)},$$

(where $k_i, m_i, n_i \geq 0$, $\delta_i = 0, 1$) form a basis of $U_q(L)$, such that the product of any two basis elements is a linear combination of basis elements with coefficients in $\mathbf{Z}[q, q^{-1}]$. The quantized enveloping algebra over $\mathbf{Z}[q, q^{-1}]$ with this basis is called the $\mathbf{Z}$-form of $U_q(L)$, and denoted by $U_{\mathbf{Z}}$. Since $U_{\mathbf{Z}}$ is defined over $\mathbf{Z}[q, q^{-1}]$ we can specialize q to any nonzero element ϵ of a field F, and obtain an algebra U_ϵ over F. In particular, if we take $\epsilon = 1$, then we obtain an algebra U_1 over $\mathbf{Q}$. Let I be the ideal of U_1 generated by $K_{\alpha_1} - 1, \ldots, K_{\alpha_l} - 1$. Then U_1/I is isomorphic to the universal enveloping algebra $U(L)$

of L. Also, the homomorphism $U_q(L) \to U(L)$ maps the basis above onto an integral basis of $U(L)$ ([Lus90]).

We call $q \in \mathbf{Q}(q)$, and $\epsilon \in F$ the quantum parameter of $U_q(L)$ and U_ϵ respectively.

108.2.6 The Canonical Basis

As in Section 108.2.4 we let U^- be the subalgebra of $U_q(L)$ generated by the F_α for $\alpha \in \Delta$. Kashiwara and Lusztig have (independently) given constructions of a basis of U^- with very nice properties, called the *canonical basis*.

Let $w_0 = s_{i_1} \cdots s_{i_t}$, and the elements F_k be as in Section 108.2.4. Then, in order to stress the dependency of the monomial

$$F_1^{(n_1)} \cdots F_t^{(n_t)}$$

on the choice of reduced expression for the longest element in $W(\Phi)$ we say that it is a w_0-monomial. The integer n_1 is called its first exponent.

Now we let $^-$ be the automorphism of U^- defined in Section 108.2.2. Elements that are invariant under $^-$ are said to be bar-invariant.

By results of Lusztig ([Lus93], Theorem 42.1.10, [Lus96], Proposition 8.2), there is a unique basis $\mathbf{B}$ of U^- with the following properties. Firstly, all elements of $\mathbf{B}$ are bar-invariant. Secondly, for any choice of reduced expression w_0 for the longest element in the Weyl group, and any element $X \in \mathbf{B}$ we have that $X = x + \sum_i \zeta_i x_i$, where x, x_i are w_0-monomials, $x \neq x_i$ for all i, and $\zeta_i \in q\mathbf{Z}[q]$. The basis $\mathbf{B}$ is called the canonical basis. If we work with a fixed reduced expression for the longest element in $W(\Phi)$, and write $X \in \mathbf{B}$ as above, then we say that x is the *principal monomial* of X.

Let $\mathcal{L}$ be the $\mathbf{Z}[q]$-lattice in U^- spanned by $\mathbf{B}$. Then $\mathcal{L}$ is also spanned by all w_0-monomials (where w_0 is a fixed reduced expression for the longest element in $W(\Phi)$).

Now let $\widetilde{w}_0$ be a second reduced expression for the longest element in $W(\Phi)$. Let x be a w_0-monomial, and let X be the element of $\mathbf{B}$ with principal monomial x. Write X as a linear combination of $\widetilde{w}_0$-monomials, and let $\widetilde{x}$ be the principal monomial of that expression. Then we write $\widetilde{x} = R_{w_0}^{\widetilde{w}_0}(x)$. Note that $x = \widetilde{x} \bmod q\mathcal{L}$.

Now let $\mathcal{B}$ be the set of all $x \bmod q\mathcal{L}$, where x runs through the set of w_0-monomials. Then $\mathcal{B}$ is a basis of the $\mathbf{Z}$-module $\mathcal{L}/q\mathcal{L}$. Moreover, $\mathcal{B}$ is independent of the choice of w_0. Let $\alpha \in \Delta$, and let $\widetilde{w}_0$ be a reduced expression for the longest element in $W(\Phi)$, starting with s_α. The Kashiwara operators $\widetilde{F}_\alpha : \mathcal{B} \to \mathcal{B}$ and $\widetilde{E}_\alpha : \mathcal{B} \to \mathcal{B} \cup \{0\}$ are defined as follows. Let $b \in \mathcal{B}$ and let x be the w_0-monomial such that $b = x \bmod q\mathcal{L}$. Set $\widetilde{x} = R_{w_0}^{\widetilde{w}_0}(x)$. Let $\widetilde{x}'$ be the $\widetilde{w}_0$-monomial constructed from $\widetilde{x}$ by increasing its first exponent by 1. Then $\widetilde{F}_\alpha(b) = R_{\widetilde{w}_0}^{w_0}(\widetilde{x}') \bmod q\mathcal{L}$. For $\widetilde{E}_\alpha$ we let $\widetilde{x}'$ be the $\widetilde{w}_0$-monomial constructed from $\widetilde{x}$ by decreasing its first exponent by 1, if this exponent is ≥ 1. Then $\widetilde{E}_\alpha(b) = R_{\widetilde{w}_0}^{w_0}(\widetilde{x}') \bmod q\mathcal{L}$. Furthermore, $\widetilde{E}_\alpha(b) = 0$ if the first exponent of $\widetilde{x}$ is 0. It can be shown that this definition does not depend on the choice of w_0, $\widetilde{w}_0$. Furthermore we have $\widetilde{F}_\alpha \widetilde{E}_\alpha(b) = b$, if $\widetilde{E}_\alpha(b) \neq 0$, and $\widetilde{E}_\alpha \widetilde{F}_\alpha(b) = b$ for all $b \in \mathcal{B}$.

Now let $V(\lambda)$ be a highest-weight module over $U_q(L)$, with highest weight λ. Let v_λ be a fixed highest weight vector. Then $\mathbf{B}_\lambda = \{X \cdot v_\lambda \mid X \in \mathbf{B}\} \setminus \{0\}$ is a basis of $V(\lambda)$,

called the *canonical basis* of $V(\lambda)$. Let $\mathcal{L}(\lambda)$ be the $\mathbf{Z}[q]$-lattice in $V(\lambda)$ spanned by $\mathbf{B}_\lambda$. We let $\mathcal{B}(\lambda)$ be the set of all $x \cdot v_\lambda \bmod q\mathcal{L}(\lambda)$, where x runs through all w_0-monomials, such that $X \cdot v_\lambda \neq 0$, where $X \in \mathbf{B}$ is the element with principal monomial x. Then the Kashiwara operators are also viewed as maps $\mathcal{B}(\lambda) \to \mathcal{B}(\lambda) \cup \{0\}$, in the following way. Let $b = x \cdot v_\lambda \bmod q\mathcal{L}(\lambda)$ be an element of $\mathcal{B}(\lambda)$, and let $b' = x \bmod q\mathcal{L}$ be the corresponding element of $\mathcal{B}$. Let y be the w_0-monomial such that $\widetilde{F}_\alpha(b') = y \bmod q\mathcal{L}$. Then $\widetilde{F}_\alpha(b) = y \cdot v_\lambda \bmod q\mathcal{L}(\lambda)$. The description of $\widetilde{E}_\alpha$ is analogous. (In [Jan96], Chapter 9 a different definition is given; however, by [Jan96], Proposition 10.9, Lemma 10.13, the two definitions agree).

The set $\mathcal{B}(\lambda)$ has $\dim V(\lambda)$ elements. We let Γ be the coloured directed graph defined as follows. The points of Γ are the elements of $\mathcal{B}(\lambda)$, and there is an arrow with colour $\alpha \in \Delta$ connecting $b, b' \in \mathcal{B}$, if $\widetilde{F}_\alpha(b) = b'$. The graph Γ is called the *crystal graph* of $V(\lambda)$.

In [Gra02] algorithms are given for computing the action of the Kashiwara operators on $\mathcal{B}$ (without computing $\mathbf{B}$ first), and for computing elements of $\mathbf{B}$.

108.2.7 The Path Model

In this section we recall some basic facts on Littelmann's path model.

From Section 108.2.2 we recall that P denotes the weight lattice. Let P_R be the vector space over R spanned by P. Let Π be the set of all piecewise linear paths $\xi : [0,1] \to P_R$, such that $\xi(0) = 0$. For $\alpha \in \Delta$ Littelmann defined *path operators* $f_\alpha, e_\alpha : \Pi \to \Pi \cup \{0\}$. Let λ be a dominant weight and let ξ_λ be the path joining λ and the origin by a straight line. Let Π_λ be the set of all nonzero $f_{\alpha_{i_1}} \cdots f_{\alpha_{i_m}}(\xi_\lambda)$ for $m \geq 0$. Then $\xi(1) \in P$ for all $\xi \in \Pi_\lambda$. Let $\mu \in P$ be a weight, and let $V(\lambda)$ be the highest-weight module over $U_q(L)$ of highest weight λ. A theorem of Littelmann states that the number of paths $\xi \in \Pi_\lambda$ such that $\xi(1) = \mu$ is equal to the dimension of the weight space of weight μ in $V(\lambda)$ ([Lit95], Theorem 9.1).

All paths appearing in Π_λ are so-called Lakshmibai–Seshadri paths (LS-paths for short). They are defined as follows. Let $\leq$ denote the Bruhat order on $W(\Phi)$. For $\mu, \nu \in W(\Phi) \cdot \lambda$ (the orbit of λ under the action of $W(\Phi)$), write $\mu \leq \nu$ if $\tau \leq \sigma$, where $\tau, \sigma \in W(\Phi)$ are the unique elements of minimal length such that $\tau(\lambda) = \mu$, $\sigma(\lambda) = \nu$. Now a rational path of shape λ is a pair $\pi = (\nu, a)$, where $\nu = (\nu_1, \ldots, \nu_s)$ is a sequence of elements of $W(\Phi) \cdot \lambda$, such that $\nu_i > \nu_{i+1}$ and $a = (a_0 = 0, a_1, \cdots, a_s = 1)$ is a sequence of rationals such that $a_i < a_{i+1}$. The path π corresponding to these sequences is given by

$$\pi(t) = \sum_{j=1}^{r-1} (a_j - a_{j-1})\nu_j + \nu_r(t - a_{r-1})$$

for $a_{r-1} \leq t \leq a_r$. Now an LS-path of shape λ is a rational path satisfying a certain integrality condition (see [Lit94], [Lit95]). We note that the path $\xi_\lambda = ((\lambda), (0,1))$ joining the origin and λ by a straight line is an LS-path of shape λ. Furthermore, all paths obtained from ξ_λ by applying the path operators are LS-paths of shape λ.

From [Lit94], [Lit95]) we transcribe the following:

(a) Let π be an LS-path. Then $f_\alpha \pi$ is an LS-path or 0; and the same holds for $e_\alpha \pi$.

(b) The action of f_α, e_α can easily be described combinatorially (see [Lit94]).

(c) The endpoint of an LS-path is an integral weight.

(d) Let $\pi = (\nu, a)$ be an LS-path. Then by $\phi(\pi)$ we denote the unique element σ of $W(\Phi)$ of shortest length such that $\sigma(\lambda) = \nu_1$.

Let λ be a dominant weight. Then we define a labeled directed graph Γ as follows. The points of Γ are the paths in Π_λ. There is an edge with label $\alpha \in \Delta$ from π_1 to π_2 if $f_\alpha \pi_1 = \pi_2$. Now by [Kas96] this graph Γ is isomorphic to the crystal graph of the highest-weight module with highest weight λ. So the path model provides an efficient way of computing the crystal graph of a highest-weight module, without constructing the module first. Also we see that $f_{\alpha_{i_1}} \cdots f_{\alpha_{i_r}} \xi_\lambda = 0$ is equivalent to $\widetilde{F}_{\alpha_{i_1}} \cdots \widetilde{F}_{\alpha_{i_r}} v_\lambda = 0$, where $v_\lambda \in V(\lambda)$ is a highest weight vector (or rather the image of it in $\mathcal{L}(\lambda)/q\mathcal{L}(\lambda)$), and the $\widetilde{F}_{\alpha_k}$ are the Kashiwara operators on $\mathcal{B}(\lambda)$ (see Section 108.2.6).

108.3 Gauss Numbers

`GaussNumber(n, v)`

Given an integer n and a ring element v, this function returns the Gauss number corresponding to n with parameter v, i.e., the element $[n]_v = v^{n-1} + v^{n-3} + \cdots + v^{-n+3} + v^{-n+1}$.

`GaussianFactorial(n, v)`

Given an integer n and a ring element v, this function returns the Gaussian factorial corresponding to n with parameter v, i.e., the element $[n]_v! = [n]_v[n-1]_v \cdots [1]_v$.

`GaussianBinomial(n, k, v)`

Given an integer n, an integer k and a ring element v, this function returns the Gaussian binomial n choose k with parameter v, i.e., the element $[n]_v!/([k]_v![n-k]_v!)$.

Example H108E1

```
> F<q>:= RationalFunctionField(Rationals());
> GaussianBinomial(5, 3, q^2);
(q^24 + q^20 + 2*q^16 + 2*q^12 + 2*q^8 + q^4 + 1)/q^12
```

108.4 Construction

QuantizedUEA(R)

QuantizedUEAlgebra(R)

QuantizedUniversalEnvelopingAlgebra(R)

This creates the quantized enveloping algebra U corresponding to the root datum R. The algebra U will be defined over the rational function field in one variable, q, over the rational numbers.

Let n and r respectively be the number of positive roots, and the rank of R. Then U has $2n+r$ generators, accessible as `U.1`, `U.2` and so on. The first n of these are printed as `F_1`, ... , `F_n`. They generate a PBW-type basis of the subalgebra U^- (cf. Section 108.2.4). The next r generators are printed as `K_1`, ... , `K_r`; together with their inverses they generate the algebra U^0. The final n generators are printed as `E_1`, ... , `E_n`. They generate a PBW-type basis of U^+.

In U we use a basis of the integral form of U (Section 108.2.5). This means that instead of F_k^s and E_k^s we use the divided powers $F_k^{(s)}$ and $E_k^{(s)}$. Furthermore, a general basis element of U^0 is a product of elements which are of the form $[K_i;t]$, or $K_i[K_i;t]$. Here $[K_i;t]$ represents the "binomial" K_i choose t as described in Section 108.2.5.

`w0` SeqEnum *Default :*

It is also possible to give a reduced expression for the longest element in the Weyl group, by setting the optional parameter `w0` equal to a sequence of indices lying between 1 and the rank of R. If we replace each index by the corresponding simple reflection, then a reduced expression for the longest element in the Weyl group has to be obtained. In that case the PBW-basis relative to that sequence will be created (and used in subsequent computations). If this parameter is not given, then the lexicographically smallest reduced expression will be used.

Example H108E2

We construct the quantum group corresponding to the root datum of type C_3.

```
> R:= RootDatum("C3");
> U:= QuantizedUEA(R);
> U.9; U.10; U.15;
F_9
K_1
E_3
> U.21*U.14*U.10*U.9*U.1;
1/q*F_1*F_9*K_1*E_2*E_9 - 1/q*F_1*F_9*K_1*E_6 + 1/q^3*F_1*K_1*[ K_3 ; 1 ]*E_2 -
    F_9*E_3*E_9 + F_9*E_8 - 1/q^2*[ K_3 ; 1 ]*E_3
```

Now we construct the same algebra, but use the PBW-basis relative to a different reduced expression of the longest element in the Weyl group.

```
> U:= QuantizedUEA(R : w0:= [2,3,1,2,3,1,2,3,1]);
```

```
> U.21*U.14*U.10*U.9*U.1;
q^2*F_1*F_9*K_1*E_2*E_9 + (q^2 - 1)/q^3*F_9*K_1*[ K_2 ; 1 ]*E_6*E_9 -
    1/q^2*F_9*K_1*K_2*E_6*E_9 - q^2*F_1*F_9*K_1*E_4 + q^2*F_1*K_1[ K_1 ; 1 ]*E_2 +
    q*F_3*K_1*E_2*E_9 + (-q^2 + 1)/q^3*F_9*K_1*[ K_2 ; 1 ]*E_7 +
    1/q^2*F_9*K_1*K_2*E_7 + (q^2 - 1)/q*K_1[ K_1 ; 1 ]*[ K_2 ; 1 ]*E_6
    - K_1[ K_1 ; 1 ]*K_2*E_6 - q*F_3*K_1*E_4 + q*F_1*E_2
```

`AssignNames(U, S)`

Assign the names in the sequence S to the generators of the algebra U.

`ChangeRing(U, R)`

Return the algebra identical to the algebra U but having coefficient ring R.

108.5 Related Structures

`CoefficientRing(U)`

This returns the ring of coefficients of the quantized enveloping algebra U.

`RootDatum(U)`

This returns the root datum corresponding to the quantized enveloping algebra U.

`PositiveRootsPerm(U)`

Given a quantized universal enveloping algebra U with root datum R returns a sequence consisting of the integers between 1 and the number of positive roots of R. If the k-th element of this sequence is m, then the generator F_k of U is of weight $-\beta_m$, where β_m is the m-th positive root of R (as returned by `PositiveRoots(R)`). (For the definition of weight of an element of U see Section 108.2.4.) Furthermore, the generator E_k is of weight β_m.

Example H108E3

```
> R:= RootDatum("D4");
> U:= QuantizedUEA(R);
> CoefficientRing(U);
Univariate rational function field over Rational Field
Variables: q
> RootDatum(U);
Adjoint root datum of type D4
> PositiveRootsPerm(U);
[ 1, 5, 2, 8, 6, 3, 12, 11, 9, 10, 7, 4 ]
```

So for instance this means that F_6 is of weight $-\beta_3$.

108.6 Operations on Elements

The generators of a quantized enveloping algebra U can be constructed by using the dot operator, e.g., `U.5`. More general elements can then be constructed using the operations of scalar multiplication, addition, and multiplication.

Note that for the generators denoted F_k and E_k we use divided powers instead of normal powers. This means for instance that $F_k^s = [s]!F_k^{(s)}$, i.e., exponentiation causes multiplication by a scalar factor.

`x + y` `x - y` `x * y` `c * x` `x * c` `x ^ n`

`U ! 0`
`Zero(U)`

The zero element of the quantized enveloping algebra U.

`U ! 1`
`One(U)`

The identity element of the quantized enveloping algebra U.

`U . i`

The i-th generator of the quantized enveloping algebra U. Let the root datum have s positive roots and rank r. If $1 \leq i \leq s$ then `U.i` is F_i. If $s+1 \leq i \leq s+r$, then `U.i` is K_j where $j = i - s$. If $s + r + 1 \leq i \leq 2s + r$ then `U.i` is E_j, where $j = i - s - r$.

`U ! r`

Returns r as an element of the quantized universal enveloping algebra U where r may be anything coercible into the coefficient ring of U or an element of another quantized enveloping algebra whose coefficients may be coerced into the coefficient ring of U.

`KBinomial(U, i, s)`
`KBinomial(K, s)`

Given a quantized enveloping algebra U corresponding to a root datum of rank r, an integer i between 1 and r, and a positive integer s, return the element $[K_i; s]$. This can be used to construct general elements in the subalgebra U^0 (cf. Section 108.2.4).

Or given an element $K = K_i$, i.e., equal to `U.(n+i)`, where n is the number of positive roots of the root datum, return $[K; s]$.

`Monomials(u)`

Given an element u of a quantized enveloping algebra, returns the sequence consisting of the monomials of u. This sequence corresponds exactly to the one returned by `Coefficients(u)`.

`Coefficients(u)`

Given an element u of a quantized enveloping algebra, returns the sequence consisting of the coefficients of the monomials that occur in u. This sequence corresponds exactly to the one returned by `Monomials(u)`.

`K ^ -1`

Given a generator K of a quantized enveloping algebra U of the form K_i, i.e., it is equal to `U.k`, for some $n+1 \leq k \leq n+r$ where U corresponds to a root datum of rank r with n positive roots, return the inverse of K.

`Degree(u, i)`

Given an element u of a quantized enveloping algebra U and an integer $1 \leq i \leq n$ or $n+r+1 \leq i \leq 2n+r$, where the root datum corresponding to U has n positive roots and rank r (i.e., `U.i` is equal to F_i or to E_k, where $k = i-n-r$), return the degree of u in the generator F_i if $1 \leq i \leq n$, otherwise return the degree of u in the generator E_k, where $k = i-n-r$.

`KDegree(m, i)`

Given a single monomial m in a quantized enveloping algebra and an integer $1 \leq i \leq r$, where r is the rank of the corresponding root datum return a tuple of 2 integers, where the first is 0 or 1, and the second is non-negative. Denote this tuple by $\langle d, k \rangle$. If $d = 0$ then the factor $[K_i; k]$ occurs in the monomial m. If $d = 1$, then the factor $K_i[K_i; k]$ occurs in the monomial m.

Example H108E4

```
> R:= RootDatum("G2");
> U:= QuantizedUEA(R);
> u:= U.10*U.7^3*U.1;
> m:= Monomials(u); m;
[
    F_1*K_1[ K_1 ; 2 ]*E_2,
    F_1*[ K_1 ; 1 ]*E_2,
    F_1*K_1*E_2,
    K_1[ K_1 ; 1 ]*E_3,
    E_3
]
> Coefficients(u);
[
    (q^6 - q^4 - q^2 + 1)/q^17,
    (q^2 - 1)/q^14,
    1/q^15,
    (-q^2 + 1)/q^9,
    -1/q^8
]
> Degree(m[1], 1);
```

```
1
> Degree(m[1], 9);
0
> Degree(m[1], 10);
1
> KDegree(m[1], 1);
<1, 2>
> U.7^-1;
(-q^2 + 1)/q*[ K_1 ; 1 ] + K_1
> U.7*U.7^-1;
1
```

108.7 Representations

In this section we describe some functions for working with left-modules over quantized enveloping algebras. For a general introduction into algebra modules in MAGMA we refer to Chapter 95.

`HighestWeightRepresentation(U, w)`

Given a quantized enveloping algebra U corresponding to a root datum of rank r and a sequence w of non-negative integers of length r, returns the irreducible representation of U with highest weight w. The object returned is a function which given an element of U computes its matrix.

`HighestWeightModule(U, w)`

Given a quantized enveloping algebra U corresponding to a root datum of rank r and a sequence w of non-negative integers of length r, returns the irreducible U-module with highest weight w. The object returned is a left module over U.

Example H108E5

```
> R:= RootDatum("G2");
> U:= QuantizedUEA(R);
> f:= HighestWeightRepresentation(U, [1,1]);
> f(U.6);
[ 0  0  0  0  0  0  0]
[ 0  0  0  0  0  0  0]
[ 0  1  0  0  0  0  0]
[ 0  0  0  0  0  0  0]
[ 0  0  0  0  0  0  0]
[ 0  0  0  0 -q  0  0]
[ 0  0  0  0  0  0  0]
> M:= HighestWeightModule(U, [1,0]);
> U.6^M.5;
(0  0  0  0  0 -q  0)
```

`WeightsAndVectors(V)`

For a module V over a quantized universal enveloping algebra this returns two sequences. The first sequence consists of the weights that occur in V. The second sequence is a sequence of sequences of elements of V, in bijection with the first sequence. The i-th element of the second sequence consists of a basis of the weight space of weight equal to the i-th weight of the first sequence.

`HighestWeightsAndVectors(V)`

This function is analogous to the previous one. Except in this case the first sequence consists of highest weights, i.e., those weights which occur as highest weights of an irreducible constituent of V. The second sequence consists of sequences that contain the corresponding highest weight vectors. So the submodules generated by the vectors in the second sequence form a direct sum decomposition of V.

`CanonicalBasis(V)`

Given a (left-) module V over a quantized universal enveloping algebra, returns the canonical basis of V. If V is not irreducible, then the union of the canonical bases of the irreducible components of V is returned.

Example H108E6

```
> U:= QuantizedUEA(RootDatum("B2"));
> V:= HighestWeightModule(U, [1,0]);
> C:= CanonicalBasis(V); C;
[
    V: (1 0 0 0 0),
    V: (0 1 0 0 0),
    V: (0 0 1 0 0),
    V: (0 0 0 1 0),
    V: (    0      0      0      0 -1/q^2)
]
```

We can compute the action of elements of U with respect to the canonical basis by using `ModuleWithBasis`:

```
> M:= ModuleWithBasis(C);
> ActionMatrix(M, U.1);
[0 0 0 0 0]
[1 0 0 0 0]
[0 0 0 0 0]
[0 0 0 0 0]
[0 0 0 1 0]
```

`TensorProduct(Q)`

Given a sequence Q of left-modules over a quantized universal enveloping algebra, returns the module M that is the tensor product of the elements of Q. It also returns a map from the Cartesian product of the elements of Q to M. This maps a tuple t to the element of M that is formed by tensoring the elements of t.

Example H108E7

```
> U:= QuantizedUEA(RootDatum("B2"));
> v1:= HighestWeightModule(U, [1,0]);
> V1:= HighestWeightModule(U, [1,0]);
> V2:= HighestWeightModule(U, [0,1]);
> W, f:= TensorProduct([V1,V2]);
> Dimension(W);
20
> HighestWeightsAndVectors(W);
[
    (1 1),
    (0 1)
]
[
    [
        W: (1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0)
    ],
    [
        W: (0   0   1   0   0   q^4   0   0   -q^7/(q^2 + 1)   0   0   0   0   0
            0   0   0   0   0   0)
    ]
]
> f(<V1.2,V2.4>);
W: (0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0)
```

So in particular we see that $V1 \otimes V2$ is the direct sum of two irreducible modules, one with highest weight $(1, 1)$, the other with highest weight $(0, 1)$. The corresponding highest-weight vectors are also given.

108.8 Hopf Algebra Structure

In this section we describe the functions for working with the Hopf algebra structure of a quantized universal enveloping algebra (cf. Section 108.2.3).

`UseTwistedHopfStructure(U, f, g)`

The Hopf algebra structure that is used by default is the one described in Section 108.2.3. As explained in that same section, it is possible to twist this by an automorphism, or an antiautomorphism.

Given a quantized universal enveloping algebra U and (anti-) automorphisms f and g of U where g is the inverse of f (this is not checked by MAGMA) set U to use the corresponding twisted Hopf algebra structure.

This command has to be given before using the Hopf algebra structure, otherwise the default structure will be used. This includes creating a tensor product.

For some (anti-) automorphisms we refer to Section 108.9.

`HasTwistedHopfStructure(U)`

This function checks whether the quantized enveloping algebra U has been set to use a twisted Hopf structure. If the first value returned by this function is `true`, then the (anti-) automorphism and its inverse are also returned.

`Counit(U)`

Returns the counit of the quantized enveloping algebra U. It is a map from U into the ground field of U.

`Antipode(U)`

Returns the antipode of the quantized enveloping algebra U. It is an antiautomorphism of U.

`Comultiplication(U, d)`

Returns the comultiplication of degree d of the quantized enveloping algebra U. This is a map from U into the d-fold tensor power of U. The comultiplication given in Section 108.2.3 is of degree 2. The comultiplications of higher degree are obtained by repeating this map. So in particular, d has to be at least 2.

An element of the d-fold tensor power of U is represented (rather primitively) by a list of d-tuples, each followed by a coefficient. The d-tuples are d-tuples of basis elements of U. The element represented by this list is the sum of the elements obtained by multiplying the i-th coefficient and the tensor product of the elements in the i-th d-tuple.

Example H108E8

```
> U:= QuantizedUEA(RootDatum("A3"));
> d:= Comultiplication(U, 2);
> d(U.1);
[*
<1, F_1>,
1,
<F_1, K_1>,
1,
<F_1, [ K_1 ; 1 ]>,
(-q^2 + 1)/q
*]
```

108.9 Automorphisms

`BarAutomorphism(U)`

For a quantized enveloping algebra U this returns the bar-automorphism of U (Section 108.2.2). The map returned by this function has its inverse stored, which can be retrieved using `Inverse`.

`AutomorphismOmega(U)`

For a quantized enveloping algebra U this returns the automorphism of U that is denoted by ω (Section 108.2.2). The map returned by this function has its inverse stored, which can be retrieved using `Inverse`.

`AntiAutomorphismTau(U)`

For a quantized enveloping algebra U this returns the anti-automorphism of U that is denoted by τ (Section 108.2.2). The map returned by this function has its inverse stored, which can be retrieved using `Inverse`.

`AutomorphismTalpha(U, k)`

Let U be a quantized enveloping algebra, and let k be an integer between 1 and the rank of the root datum. Then this function returns the automorphism T_{α_k} of U, corresponding to the k-th simple root (Section 108.2.4). The map returned by this function has its inverse stored, which can be retrieved using `Inverse`.

`DiagramAutomorphism(U, p)`

`GraphAutomorphism(U, p)`

Let U be a quantized enveloping algebra, and let p be a permutation of $\{1, \ldots, r\}$, where r is the rank of the root datum. Here p must represent a diagram automorphism of the root datum (i.e., it leaves the Dynkin diagram invariant). Then this function returns the corresponding automorphism of U (see Section 108.2.2). The map returned by this function has its inverse stored, which can be retrieved using `Inverse`.

Example H108E9

```
> R:= RootDatum("G2");
> U:= QuantizedUEA(R);
> b:= BarAutomorphism(U);
> b(U.3);
(q^10 - q^6 - q^4 + 1)/q^4*F_1^(2)*F_6 + (q^4 - 1)/q^2*F_1*F_5 + F_3
```

A known result states that $T_{\alpha_r}^{-1} = \tau \circ T_{\alpha_r} \circ \tau$. We check that for the quantum group of type C_3, and the third simple root.

```
> U:= QuantizedUEA(RootDatum("C3"));
> t:= AntiAutomorphismTau(U);
> T:= AutomorphismTalpha(U, 3);
> Ti:= Inverse(T);
> f:= t*T*t;
> &and[ Ti(U.i) eq f(U.i) : i in [1..21] ];
true
```

A diagram automorphism maps the canonical basis into itself. We check that for the set of elements of the canonical basis of the quantized enveloping algebra of type D_4 of weight $\alpha_1 + 3\alpha_2 + 2\alpha_3 + 2\alpha_4$. (Here α_i is the i-th simple root.) The chosen diagram automorphism maps this weight to $2\alpha_1 + 3\alpha_2 + \alpha_3 + 2\alpha_4$. Therefore we also compute the elements of the canonical basis of that weight.

```
> U:= QuantizedUEA(RootDatum("D4"));
> p:= SymmetricGroup(4)!(1,3,4);
> d:= DiagramAutomorphism(U, p);
> e1:= CanonicalElements(U, [1,3,2,2]);
> e2:= CanonicalElements(U, [2,3,1,2]);
> &and[ d(x) in e2 : x in e1 ];
true
```

108.10 Kashiwara Operators

`Falpha(m, i)`

Given a monomial m in U^- for some quantized enveloping algebra U, i.e., m must be a monomial in the first n generators of U, where n is the number of positive roots of the corresponding root datum, returns another monomial in the negative part of U that is obtained by applying the i-th Kashiwara operator $\tilde{F}_i$ to m (see Section 108.2.6). Here i must lie between 1 and the rank of the root datum.

`Ealpha(m, i)`

Given a monomial m in U^- for some quantized enveloping algebra U, i.e., m must be a monomial in the first n generators of U, where n is the number of positive roots of the corresponding root datum, return $\tilde{E}_i(m)$ (see Section 108.2.6) if the i-th Kashiwara operator $\tilde{E}_i$ is applicable to m. Otherwise the zero element of U is returned. Here i must lie between 1 and the rank of the root datum.

Example H108E10

```
> R:= RootDatum("F4");
> U:= QuantizedUEA(R);
> m:= U.1*U.5*U.10*U.18*U.24;
> m;
F_1*F_5*F_10*F_18*F_24
> Falpha(m, 3);
F_1*F_6*F_7*F_10*F_18*F_24
> Ealpha(m, 4);
F_1*F_4*F_5*F_7*F_9*F_18*F_24
> Ealpha(m, 2);
0
```

108.11 The Path Model

In this section we describe functions for working with Littelmann's path model (cf. Section 108.2.7). A special role is played by the zero path. The path operators cannot be applied to the zero path. However, on some occasions they do produce the zero path.

`DominantLSPath(R, hw)`

Given a root datum R and a sequence hw of non-negative integers returns the path that is the straight line from the origin to hw.

`Falpha(p, i)`

Given a (non-zero) path p and an integer i between 1 and the rank of the root datum returns the result of applying the path operator f_{α_i} to p (where α_i is the i-th simple root).

`Ealpha(p, i)`

Given a (non-zero) path p and an integer i between 1 and the rank of the root datum returns the result of applying the path operator e_{α_i} to p (where α_i is the i-th simple root).

`WeightSequence(p)`

For a path p this returns the sequence of weights that, along with the sequence of rational numbers, defines the path (cf. Section 108.2.7).

`RationalSequence(p)`

For a path p this returns the sequence of rational numbers that, along with the sequence of weights, defines the path (cf. Section 108.2.7).

`EndpointWeight(p)`

Returns the weight which is the end point of the path p.

`Shape(p)`

Returns the weight which is the shape of the path p.

`WeylWord(p)`

Returns a reduced expression for the element σ of the Weyl group, of shortest length such that $\sigma(\lambda) = \nu_1$, where λ is the shape of the path p, and ν_1 is the first weight in the sequence `WeightSequence(p)`. The reduced expression is represented as a sequence of integers between 1 and the rank of the root datum. In this sequence the index i represents the i-th simple reflection.

`IsZero(p)`

Returns `true` if the path p is the zero path, `false` otherwise.

`p1 eq p2`

Returns `true` if the paths $p1$ and $p2$ are equal, `false` otherwise.

Example H108E11

```
> R:= RootDatum("B2");
> p:= DominantLSPath(R, [ 2, 3 ]);
> p;
LS-path of shape (2 3) ending in (2 3)
> Falpha(p, 1);
LS-path of shape (2 3) ending in (0 5)
> Ealpha(Falpha(p, 1), 1);
LS-path of shape (2 3) ending in (2 3)
> p1:= Falpha(Falpha(Falpha(p, 1), 2), 1);
> p1;
LS-path of shape (2 3) ending in (-1  5)
> WeightSequence(p1);
[
    (5 -7),
    (-2  7)
]
> RationalSequence(p1);
[ 0, 1/7, 1 ]
> WeylWord(p1);
[ 2, 1 ]
```

So $s_2 s_1(2,3) = (5,-7)$.

`CrystalGraph(R, hw)`

For a root datum R and a sequence of non-negative integers hw (of length equal to the rank of the root datum), this function returns the corresponding crystal graph G, along with a sequence of paths. The graph G is a directed labelled graph. The labels on the edges are integers between 1 and the rank of the root system. If there is an edge from i to j with label s, then $f_{\alpha_s}(p_i) = p_j$, where p_i, p_j are the i-th and j-th elements of the sequence of paths returned by this function (and f_{α_s} is the root operator corresponding to the s-th simple root). In other words, the i-th path is the i-th point of the graph G.

Example H108E12

```
> R:= RootDatum("G2");
> G, pp:= CrystalGraph(R, [0,1]);
> G;
Digraph
Vertex  Neighbours
1       2 ;
2       3 ;
3       4 ;
4       5 6 ;
```

```
5       7 ;
6       8 ;
7       9 ;
8       10 ;
9       11 ;
10      11 ;
11      12 ;
12      13 ;
13      14 ;
14      ;
> e:= Edges(G);
> e[10];
[9, 11]
> Label(e[10]);
1
> Falpha(pp[9], 1) eq pp[11];
true
```

108.12 Elements of the Canonical Basis

`CanonicalElements(U, w)`

Given a quantized enveloping algebra U, corresponding to a root datum R of rank r and a sequence w of non-negative integers of length r returns the sequence consisting of the elements of the canonical basis of the negative part of U that are of weight ν, where ν denotes the linear combination of the simple roots of R defined by w (i.e., $\nu = w[1]\alpha_1 + \cdots + w[r]\alpha_r$ and $\alpha_1, \ldots, \alpha_r$ denote the simple roots of R).

Example H108E13

```
> R:= RootDatum("F4");
> U:= QuantizedUEA(R);
> c:= CanonicalElements(U, [1,2,1,1]); c;
[
    F_1*F_3^(2)*F_9*F_24,
    q*F_1*F_3^(2)*F_9*F_24 + F_1*F_3^(2)*F_23,
    q^4*F_1*F_3^(2)*F_9*F_24 + F_1*F_3*F_7*F_24,
    q^5*F_1*F_3^(2)*F_9*F_24 + q^4*F_1*F_3^(2)*F_23 + q*F_1*F_3*F_7*F_24 +
        F_1*F_3*F_21,
    q^4*F_1*F_3^(2)*F_9*F_24 + F_2*F_3*F_9*F_24,
    q^5*F_1*F_3^(2)*F_9*F_24 + q^4*F_1*F_3^(2)*F_23 + q*F_2*F_3*F_9*F_24 +
        F_2*F_3*F_23,
    (q^6 + q^2)*F_1*F_3^(2)*F_9*F_24 + q^2*F_1*F_3*F_7*F_24 +
        q^2*F_2*F_3*F_9*F_24 + F_2*F_7*F_24,
    (q^7 + q^3)*F_1*F_3^(2)*F_9*F_24 + (q^6 + q^2)*F_1*F_3^(2)*F_23 +
        q^3*F_1*F_3*F_7*F_24 + q^3*F_2*F_3*F_9*F_24 + q^2*F_1*F_3*F_21 +
```

```
        q^2*F_2*F_3*F_23 + q*F_2*F_7*F_24 + F_2*F_21,
    q^8*F_1*F_3^(2)*F_9*F_24 + q^4*F_1*F_3*F_7*F_24 + q^4*F_2*F_3*F_9*F_24 +
        q^2*F_2*F_7*F_24 + F_3*F_4*F_24,
    q^9*F_1*F_3^(2)*F_9*F_24 + q^8*F_1*F_3^(2)*F_23 + q^5*F_1*F_3*F_7*F_24 +
        q^5*F_2*F_3*F_9*F_24 + q^4*F_1*F_3*F_21 + q^4*F_2*F_3*F_23 +
        q^3*F_2*F_7*F_24 + q*F_3*F_4*F_24 + q^2*F_2*F_21 + F_3*F_18
]
> b:= BarAutomorphism(U);
> [ b(u) eq u : u in c ];
[ true, true, true, true, true, true, true, true, true, true ]
```

All elements of the canonical basis are invariant under the bar-automorphism.

Example H108E14

In the next example we show how to use the crystal graph to determine whether an element of the canonical basis acting on the highest weight vector of an irreducible module gives zero or not.

```
> U:= QuantizedUEA(RootDatum("A2"));
> G, p:= CrystalGraph(RootDatum(U), [1,1]);
> e:= Edges(G);
> for edge in e do
> print edge, Label(edge);
> end for;
[1, 2] 1
[1, 3] 2
[2, 4] 2
[3, 5] 1
[4, 6] 2
[5, 7] 1
[6, 8] 1
[7, 8] 2
```

We see that $f_{\alpha_1} f_{\alpha_2} f_{\alpha_2} f_{\alpha_1}(p_1) = p_8$ (where p_i is the i-th path in p). We apply the same sequence of Kashiwara operators to the identity element of U.

```
> Falpha(Falpha(Falpha(Falpha(One(U), 1), 2), 2), 1);
F_1*F_2*F_3
```

Now the element of the canonical basis with this principal monomial (see Section 108.2.6) acting on the highest weight vector of the irreducible module with highest weight `[1,1]` gives a non-zero result. The weight of this monomial is $2\alpha_1 + 2\alpha_2$. All other elements of the canonical basis of this weight give zero, as there is only one point of the crystal graph that gives a monomial of this weight.

```
> V:= HighestWeightModule(U, [1,1]);
> ce:= CanonicalElements(U, [2,2]);
> ce;
[
    F_1^(2)*F_3^(2),
    (q^3 + q)*F_1^(2)*F_3^(2) + F_1*F_2*F_3,
    q^4*F_1^(2)*F_3^(2) + q*F_1*F_2*F_3 + F_2^(2)
```

```
]
> v0:= V.1;
> ce[2]^v0;
V: (  0    0    0    0    0    0    0 -1/q)
> ce[1]^v0;
V: (0 0 0 0 0 0 0 0)
> ce[3]^v0;
V: (0 0 0 0 0 0 0 0)
```

108.13 Homomorphisms to the Universal Enveloping Algebra

`QUAToIntegralUEAMap(U)`

Given a quantized enveloping algebra U returns the map from U onto the integral form of the universal enveloping algebra of the corresponding Lie algebra (cf. Section 108.2.5). We refer to Section 106.17 for an account of universal enveloping algebras in MAGMA.

Example H108E15

```
> U:= QuantizedUEA(RootDatum("C3"));
> f:= QUAToIntegralUEAMap(U);
> p:= CanonicalElements(U, [1,2,1]);
> [ f(u) : u in p ];
[
    y_1*y_2^(2)*y_3,
    2*y_1*y_2^(2)*y_3 + y_1*y_2*y_5,
    y_1*y_2^(2)*y_3 + y_1*y_2*y_5 + y_1*y_7,
    y_1*y_2^(2)*y_3 + y_2*y_3*y_4 - y_2*y_6,
    2*y_1*y_2^(2)*y_3 + y_1*y_2*y_5 + y_2*y_3*y_4 - y_2*y_6 + y_4*y_5,
    2*y_1*y_2^(2)*y_3 + y_1*y_2*y_5 + 2*y_2*y_3*y_4 - y_2*y_6 + y_4*y_5,
    y_1*y_2^(2)*y_3 + y_1*y_2*y_5 + y_2*y_3*y_4 + y_1*y_7 + y_4*y_5 + y_8
]
```

So this allows one to construct elements of the canonical basis of a universal enveloping algebra (of a semisimple Lie algebra).

108.14 Bibliography

[**Gra04**] W. A. de Graaf. Five constructions of representations of quantum groups. *Note di Matematica*, 22(1):27–48, 2003/04.

[**Gra01**] W. A. de Graaf. Computing with quantized enveloping algebras: PBW-type bases, highest-weight modules, R-matrices. *J. Symbolic Comput.*, 32(5):475–490, 2001.

[**Gra02**] W. A. de Graaf. Constructing canonical bases of quantized enveloping algebras. *Experimental Mathematics*, 11(2):161–170, 2002.

[**Jan96**] J. C. Jantzen. *Lectures on Quantum Groups*, volume 6 of *Graduate Studies in Mathematics*. American Mathematical Society, 1996.

[**Kas96**] M. Kashiwara. Similarity of crystal bases. In *Lie algebras and their representations (Seoul, 1995)*, pages 177–186. Amer. Math. Soc., Providence, RI, 1996.

[**Lit94**] P. Littelmann. A Littlewood-Richardson rule for symmetrizable Kac-Moody algebras. *Invent. Math.*, 116(1-3):329–346, 1994.

[**Lit95**] P. Littelmann. Paths and root operators in representation theory. *Ann. of Math. (2)*, 142(3):499–525, 1995.

[**Lus90**] G. Lusztig. Quantum groups at roots of 1. *Geom. Dedicata*, 35(1-3):89–113, 1990.

[**Lus93**] G. Lusztig. *Introduction to quantum groups*. Birkhäuser Boston Inc., Boston, MA, 1993.

[**Lus96**] G. Lusztig. Braid group action and canonical bases. *Adv. Math.*, 122(2):237–261, 1996.

109 GROUPS OF LIE TYPE

109.1 Introduction 3337

109.1.1 The Steinberg Presentation . . . 3337

109.1.2 Bruhat Normalisation 3337

109.1.3 Twisted Groups of Lie type . . . 3338

109.2 Constructing Groups of Lie Type 3338

109.2.1 Split Groups 3338

GroupOfLieType(N, k) 3339
GroupOfLieType(N, q) 3339
GroupOfLieType(W, k) 3339
GroupOfLieType(W, q) 3339
GroupOfLieType(R, k) 3340
GroupOfLieType(R, q) 3340
GroupOfLieType(C, k) 3340
GroupOfLieType(D, k) 3340
GroupOfLieType(C, q) 3340
GroupOfLieType(D, q) 3340
SimpleGroupOfLieType(X, n, k) 3340
SimpleGroupOfLieType(X, n, q) 3341
GroupOfLieType(L) 3341
IsNormalising(G) 3341

109.2.2 Galois Cohomology 3341

GammaGroup(k, G) 3341
GammaGroup(k, A) 3342
ActingGroup(G) 3342
ActingGroup(A) 3342
ExtendGaloisCocycle(c) 3342
GaloisCohomology(A) 3342
IsInTwistedForm(x, c) 3342

109.2.3 Twisted Groups 3345

TwistedGroupOfLieType(c) 3345
TwistedGroupOfLieType(R, k, K) 3345
TwistedGroupOfLieType(R, q, r) 3345
BaseRing(G) 3345
CoefficientRing(G) 3345
DefRing(G) 3345
UntwistedOvergroup(G) 3345
RelativeRootElement(G,delta,t) 3346

109.3 Operations on Groups of Lie Type 3347

eq 3347
subset 3347
IsAlgebraicallyIsomorphic(G, H) 3347
IsIsogenous(G, H) 3347
IsCartanEquivalent(G, H) 3347
BaseRing(G) 3347
CoefficientRing(G) 3347
BaseExtend(G, K) 3347
ChangeRing(G, K) 3347
Generators(G) 3347
NumberOfGenerators(G) 3348
Ngens(G) 3348
AlgebraicGenerators(G) 3348
NumberOfAlgebraicGenerators(G) 3348
Nalggens(G) 3348
Order(G) 3348
3348
FactoredOrder(G) 3348
Dimension(G) 3348
CartanName(G) 3349
RootDatum(G) 3349
DynkinDiagram(G) 3349
CoxeterDiagram(G) 3349
CoxeterMatrix(G) 3349
CoxeterGraph(G) 3349
CartanMatrix(G) 3349
DynkinDigraph(G) 3349
Rank(G) 3349
ReductiveRank(G) 3349
SemisimpleRank(G) 3350
CoxeterNumber(G) 3350
WeylGroup(G) 3350
WeylGroup(GrpPermCox, G) 3350
WeylGroup(GrpFPCox, G) 3350
WeylGroup(GrpMat, G) 3350
FundamentalGroup(G) 3350
IsogenyGroup(G) 3350
CoisogenyGroup(G) 3350

109.4 Properties of Groups of Lie Type 3351

IsFinite(G) 3351
IsAbelian(G) 3351
IsSimple(G) 3351
IsSimplyLaced(G) 3351
IsSemisimple(G) 3351
IsAdjoint(G) 3351
IsWeaklyAdjoint(G) 3351
IsSimplyConnected(G) 3351
IsWeaklySimplyConnected(G) 3351
IsSplit(G) 3351
IsTwisted(G) 3352

109.5 Constructing Elements . . . 3352

elt< > 3352
Identity(G) 3352
Id(G) 3352
! 3352
elt< > 3352
TorusTerm(G, r, t) 3353
CoxeterElement(G) 3353
Random(G) 3353
Eltlist(g) 3353
CentrePolynomials(G) 3353
CenterPolynomials(G) 3353

109.6 Operations on Elements . . 3354
109.6.1 Basic Operations 3354
* 3354
^ 3354
Inverse(G) 3354
^ 3354
^ 3354
(g, h) 3355
Commutator(g, h) 3355
Normalise(~g) 3355
Normalize(~g) 3355
Normalise(g) 3355
Normalize(g) 3355
109.6.2 Decompositions 3355
Bruhat(g) 3355
Multiplicative JordanDecomposition(x) 3356
109.6.3 Conjugacy and Cohomology . . 3356
ConjugateIntoTorus(g) 3356
ConjugateIntoBorel(g) 3356
Lang(c, q) 3356

109.7 Properties of Elements . . . 3357
IsSemisimple(x) 3357
IsUnipotent(x) 3357
IsCentral(x) 3357

109.8 Roots, Coroots and Weights 3357
109.8.1 Accessing Roots and Coroots . . 3357
RootSpace(G) 3357
CorootSpace(G) 3357
SimpleRoots(G) 3357
SimpleCoroots(G) 3357
NumberOfPositiveRoots(G) 3357
NumPosRoots(G) 3357
Roots(G) 3358
Coroots(G) 3358
PositiveRoots(G) 3358
PositiveCoroots(G) 3358
Root(G, r) 3358
Coroot(G, r) 3358
RootPosition(G, v) 3358
CorootPosition(G, v) 3358
HighestRoot(G) 3359
HighestLongRoot(G) 3359
HighestShortRoot(G) 3359
109.8.2 Reflections 3360
Reflections(G) 3360
Reflection(G, r) 3360
109.8.3 Operations and Properties for Root and Coroot Indices 3360
RootHeight(G, r) 3360
CorootHeight(G, r) 3360
RootNorms(G) 3360
CorootNorms(G) 3360
RootNorm(G, r) 3360
CorootNorm(G, r) 3360
IsLongRoot(G, r) 3360
IsShortRoot(G, r) 3360
AdditiveOrder(G) 3360
109.8.4 Weights 3361
WeightLattice(G) 3361
CoweightLattice(G) 3361
FundamentalWeights(G) 3361
FundamentalCoweights(G) 3361
DominantWeight(G, v) 3361

109.9 Building Groups of Lie Type 3361
SubsystemSubgroup(G, a) 3361
SubsystemSubgroup(G, s) 3361
DirectProduct(G1, G2) 3362
Dual(G) 3362
SolubleRadical(G) 3362
StandardMaximalTorus(G) 3362

109.10 Automorphisms 3363
109.10.1 Basic Functionality 3363
AutomorphismGroup(G) 3363
IdentityAutomorphism(G) 3363
One(A) 3363
Id(A) 3363
Mapping(a) 3363
Automorphism(m) 3363
* 3363
^ 3363
~ 3363
Domain(A) 3364
Codomain(A) 3364
Domain(h) 3364
Codomain(h) 3364
109.10.2 Constructing Special Automorphisms 3364
InnerAutomorphism(G, x) 3364
DiagonalAutomorphism(G, v) 3364
GraphAutomorphism(G, p) 3364
DiagramAutomorphism(G, p) 3364
FieldAutomorphism(G, sigma) 3364
RandomAutomorphism(G) 3364
Random(A) 3364
DualityAutomorphism(G) 3365
FrobeniusMap(G,q) 3365
109.10.3 Operations and Properties of Automorphisms 3365
DecomposeAutomorphism(h) 3365
IsAlgebraic(h) 3365

109.11 Algebraic Homomorphisms . 3366
GroupOfLieTypeHomomorphism(phi, k) 3366

109.12 Twisted Tori 3366
TwistedTorusOrder(R, w) 3366
TwistedToriOrders(G) 3366
TwistedToriOrders(R) 3366

`TwistedTorus(G, w)` 3367
`TwistedTori(G)` 3367
109.13 Sylow Subgroups 3368
`PrintSylowSubgroupStructure(G)` 3368
`SylowSubgroup(G, p)` 3368
109.14 Representations 3369
`StandardRepresentation(G)` 3369
`AdjointRepresentation(G)` 3370
`LieAlgebra(G)` 3370
`HighestWeightRepresentation(G, v)` 3370
`GeneralisedRowReduction(`ρ`)` 3371
`RowReductionHomomorphism(`ρ`)` 3371
`Inverse(`ρ`)` 3371
109.15 Bibliography 3371

Chapter 109
GROUPS OF LIE TYPE

109.1 Introduction

This chapter describes MAGMA functions for computing with groups of Lie type. These functions are based on [CMT04] for split types, and [Hal05] for twisted types.

Given an extended root datum and ring with a Γ-action, a group of Lie type can be constructed in MAGMA. Such groups include reductive Lie groups (when the ring is $\mathbf{R}$ or $\mathbf{C}$), reductive algebraic groups (when the ring is an algebraically closed field), and finite groups of Lie type (when the ring is a finite field).

109.1.1 The Steinberg Presentation

The approach to computation in split groups of Lie type described here is based on the Steinberg presentation[Ste62] Let G be a split group of Lie type with root datum R over the ring k. Suppose the roots of R are $\alpha_1, \ldots, \alpha_{2N}$ ordered as in Section 103.5 and n is the rank of R. Then G contains *root elements* $x_r(t) = x_{\alpha_r}(t)$ for t in k. If R is semisimple, the root elements generate G. In the general case, it is necessary to introduce extra torus elements. Let $Y = \mathbf{Z}^d$ be the coroot space of the root datum. The torus is taken to be the abelian group $Y \otimes k^\times$, represented as the set of vectors in k^d with each component invertible, and multiplication is performed componentwise. The Weyl group of G is just the Coxeter group of the root datum RD. Redundant generators n_r are also included, corresponding to the generators s_r of the Weyl group.

Since the generating set is parametrised by field elements it is generally not possible to define G within the category of finitely presented groups `GrpFP`, so groups of Lie type form their own category, `GrpLie`.

Note that groups of Lie type in MAGMA are designed primarily for fields whose elements are exact. While it is possible to define these groups over real and complex fields (Chapter 25), no attempt has been made to control rounding error in this case.

109.1.2 Bruhat Normalisation

The Bruhat decomposition [Car93, Chapter 2] gives us a useful normal form for elements of a split group of Lie type defined over a field k. Every $g \in G$ can be written in the form $uh\dot{w}u'$ where

1. u is a unipotent element written in the form $\prod_{r=1}^{N} x_r(t_r)$;
2. h is a torus element represented as an element of R^d with each entry invertible;
3. $\dot{w} = \dot{s}_{r_1} \cdots \dot{s}_{r_k}$ where $s_{r_1} \cdots s_{r_k}$ is a reduced word for w in the Weyl group.
4. $u' = \prod_{r \in \Phi_w^+} x_r(t'_r)$ where $\Phi_w^+ = \{r \mid \alpha_r \text{ and } \alpha_r w \text{ are positive}\}$ and the terms are in the usual order.

109.1.3 Twisted Groups of Lie type

Let G be a connected reductive linear algebraic group defined over the field k. We say that H is a *form* of G if there is a $\bar{k}$-isomorphism between G and H, where $\bar{k}$ the algebraic closure of k. If some maximal torus of $G(\bar{k})$ is a k-split torus, we say that G is *split*, otherwise G is *twisted*. If G has a Borel subgroup defined over k, we say that G is *quasisplit*. There is a unique split form of every reductive linear algebraic group.

The group $\Gamma := \mathrm{Gal}(\bar{k} : k)$ acts on G in the usual way and G is a Γ-group in the sense of the Section 73.10. The group $\mathrm{Aut}(G)$ of algebraic automorphisms of G is also a Γ-group. The twisted forms of G are in one-to-one correspondence with the 1-cocycles of Γ on $\mathrm{Aut}(G)$ and the forms are conjugate if and only if the cocycles are cohomologous. For practical purposes it is sufficient to compute the cohomology of $\Gamma = \mathrm{Gal}(K : k)$ on $\mathrm{Aut}_K(G)$ for some finite Galois extension K of k, where $\mathrm{Aut}_K(G)$ is the group of K-algebraic automorphisms of G.

The action of Γ on G induces an action on the root datum of G, and so we get an extended root datum. If G is quasisplit, then it is determined by the extended root datum and the action of Γ on K. In general, a cocycle is required to fully determine G.

109.2 Constructing Groups of Lie Type

109.2.1 Split Groups

The following optional parameters are common to most of the intrinsics described in this section:

`Normalising` BoolElt *Default :* `true`

The flag `Normalising` determines whether elements will be automatically converted to Bruhat form. This flag is automatically set to `false` if the group is defined over a nonfield.

`Isogeny` BoolElt *Default :* "*Ad*"

`Signs` Any *Default :* 1

The optional parameters `Isogeny` and `Signs` can take the values described in Section 103.2.

`Method` MonStgElt *Default :* "*Default*"

The method to be used for operations with unipotent elements. See [CHM08] for more details on the algorithms. Possible values are

- `"CollectionFromLeft"` uses collection from left.
- `"CollectionFromOutside"` uses collection from outside.
- `"Classical"` uses formulas for classical types [CHM08]. This is only available for groups defined over a sparse (classical) root datum.
- `"Collection"` will choose the best of the above methods automatically.
- `"SymbolicFromLeft"` uses Hall polynomials, which are computed using collection from left.

- `"SymbolicFromOutside"` uses Hall polynomials, which are computed using collection from outside.
- `"SymbolicClassical"` uses Hall polynomials, which are computed by formulas. This is only available for groups defined over a sparse (classical) root datum.
- `"Symbolic"` will choose the best symbolic method automatically.
- `"Default"` will choose the best of all above methods automatically.

`GroupOfLieType(N, k)`

`Isogeny`	BoolElt	*Default :* "*Ad*"
`Signs`	Any	*Default :* 1
`Normalising`	BoolElt	*Default :* `true`
`Method`	MonStgElt	*Default :* "*Default*"

Construct the group of Lie type with Cartan name given by the string N (see Section 101.6) over the ring k.

`GroupOfLieType(N, q)`

`Isogeny`	BoolElt	*Default :* "*Ad*"
`Signs`	Any	*Default :* 1
`Normalising`	BoolElt	*Default :* `true`
`Method`	MonStgElt	*Default :* "*Default*"

Construct the group of Lie type with Cartan name given by the string N (see Section 101.6) over the finite field of order q.

`GroupOfLieType(W, k)`

`Normalising`	BoolElt	*Default :* `true`
`Method`	MonStgElt	*Default :* "*Default*"

Construct the group of Lie type with Weyl group W over the ring k. The group W must be a finite Coxeter group, given either as a permutation group or as a reflection group.

`GroupOfLieType(W, q)`

`Normalising`	BoolElt	*Default :* `true`
`Method`	MonStgElt	*Default :* "*Default*"

Construct the group of Lie type with Weyl group W over the finite field of order q. The group W must be a finite Coxeter group, given either as a permutation group or as a reflection group.

`GroupOfLieType(R, k)`

`Normalising`	BoolElt	*Default :* `true`
`Method`	MonStgElt	*Default :* "*Default*"

Construct the group of Lie type with root datum R over the ring k.

`GroupOfLieType(R, q)`

`Normalising`	BoolElt	*Default :* `true`
`Method`	MonStgElt	*Default :* "*Default*"

Construct the group of Lie type with root datum R over the finite field of order q.

`GroupOfLieType(C, k)`

`GroupOfLieType(D, k)`

`Isogeny`	BoolElt	*Default :* "*Ad*"
`Signs`	Any	*Default :* 1
`Normalising`	BoolElt	*Default :* `true`
`Method`	MonStgElt	*Default :* "*Default*"

Construct the group of Lie type with Cartan matrix C or Dynkin digraph D, over the ring k.

`GroupOfLieType(C, q)`

`GroupOfLieType(D, q)`

`Isogeny`	BoolElt	*Default :* "*Ad*"
`Signs`	Any	*Default :* 1
`Normalising`	BoolElt	*Default :* `true`
`Method`	MonStgElt	*Default :* "*Default*"

Construct the group of Lie type with Cartan matrix C or Dynkin digraph D, over the finite field of order q.

`SimpleGroupOfLieType(X, n, k)`

`Isogeny`	BoolElt	*Default :* "*Ad*"
`Signs`	Any	*Default :* 1
`Normalising`	BoolElt	*Default :* `true`
`Method`	MonStgElt	*Default :* "*Default*"

Construct the simple group of Lie type with Cartan name X_n over the ring k, where the Cartan name is given by the string X and integer n (see also Section 101.6).

`SimpleGroupOfLieType(X, n, q)`

`Isogeny`	BoolElt	*Default :* "*Ad*"
`Signs`	Any	*Default :* 1
`Normalising`	BoolElt	*Default :* `true`
`Method`	MonStgElt	*Default :* "*Default*"

Construct the simple group of Lie type with name X_n over the finite field of order q, where the Cartan name is given by the string X and integer n (see also Section 101.6).

`GroupOfLieType(L)`

The group of Lie type corresponding to the Lie algebra L. The Lie algebra must be the algebraic (i.e., it must correspond to some group), and Magma must be able to determine that it is algebraic.

`IsNormalising(G)`

Returns the value of the flag `Normalising` of the group of Lie type G.

Example H109E1

```
> G := GroupOfLieType("E8", 2);
> G;
G: Group of Lie type E8 over Finite field of size 2
```

109.2.2 Galois Cohomology

If G is a linear algebraic group defined over the field k and L is the algebraic closure of k, then the group $\Gamma := \mathrm{Gal}(L : k)$ acts on G in the usual way and G becomes a Γ-group in the sense of the Section 73.10 and $\mathrm{Aut}(G)$, the group of algebraic automorphisms of G also becomes a Γ-group.

Now the twisted forms of G are in one-to-one correspondence to the 1-cocycles of Γ on $\mathrm{Aut}(G)$ and the forms are conjugate if and only if the cocycles are cohomologous.

For practical purposes it is sufficient to compute the cohomology of $\mathrm{Gal}(K : k)$ on $\mathrm{Aut}_K(G)$ for some finite Galois field extension of k, where $\mathrm{Aut}_K(G)$ is the group of K-algebraic automorphisms of G.

These functions are based on [Hal05].

`GammaGroup(k, G)`

Returns the group of Lie type G as a Γ-group with $\Gamma = \mathrm{Gal}(K : k)$, where K is the base field of G. The field k must be a subfield of K.

`GammaGroup(k, A)`

Returns the group $A = \mathrm{Aut}_K(G)$ of automorphisms of the group of Lie type G as a Γ-group with $\Gamma = \mathrm{Gal}(K : k)$, where K is the base field of G. The field k must be a subfield of K.

`ActingGroup(G)`

`ActingGroup(A)`

Given the group of Lie type G or the group A of its automorphisms as a Γ-group, return $\Gamma = \mathrm{Gal}(K : k)$ together with the map m from the abstract Galois group Γ into the set of field automorphisms, such that $m(\gamma)$ is the actual field automorphism for every $\gamma \in \Gamma$.

`ExtendGaloisCocycle(c)`

`GBAl`	MonStgElt	*Default :* "*Walk*"
`Printeqs`	BoolElt	*Default :* `false`

The analogue to `ExtendCocycle`. Given a cocycle c in $H^1(\Gamma, A/A_0)$, where $A = \mathrm{Aut}_K(G)$ and $\Gamma = \mathrm{Gal}(K : k)$, extend the cocycle to a cocycle in $H^1(\Gamma, A)$. The optional parameter `GBAl` can be used to set the algorithm used for computing the Gröbner bases. The parameter `Printeqs` may be used to print out the polynomials whose Gröbner bases are computed. The current implementation only works for finite fields.

`GaloisCohomology(A)`

`GBAl`	MonStgElt	*Default :* "*Walk*"
`Printeqs`	BoolElt	*Default :* `false`
`Recompute`	BoolElt	*Default :* `false`

Computes the Galois cohomology $H^1(\Gamma, \mathrm{Aut}_K(G))$, where A is the automorphism group of G as a Γ-group returned by `GammaGroup` and $\Gamma = \mathrm{Gal}(K : k)$. The optional parameter `GBAl` can be used to set the algorithm used for computing the Gröbner bases. The parameter `Printeqs` may be used to print out the polynomials whose Gröbner bases are computed. And `Recompute` may be used to recompute the Galois cohomology. The current implementation only works for finite fields.

`IsInTwistedForm(x, c)`

Returns `true` if and only if the element x of a group of Lie type is contained in the twisted form of its parent defined by the cocycle c.

Example H109E2

Compute the Galois cohomology of $A_3(5^2)$:

```
> q := 5;
> k := GF(q);
> K := GF(q^2);
>
> G := GroupOfLieType( "A3", K : Isogeny:="SC" );
> A := AutomorphismGroup(G);
>
> AGRP := GammaGroup( k, A );
> Gamma,m := ActingGroup(AGRP);
> Gamma;
Symmetric group Gamma acting on a set of cardinality 2
Order = 2
    (1, 2)
> m;
Mapping from: GrpPerm: Gamma to Set of all maps from GF(5^2) to GF(5^2)
given by a rule [no inverse]
> action  := GammaAction(AGRP);
>
> time GaloisCohomology(AGRP);
[
    [
        One-Cocycle
        defined by [
        Automorphism of $: Group of Lie type A3 over Finite field of size 5^2
        given by: Mapping from: $: Group of Lie type  to $: Group of Lie type
        Composition of Mapping from: $: Group of Lie type  to $: Group of
        Lie type  given by a rule and
        Mapping from: $: Group of Lie type  to $: Group of Lie type
        given by a rule
        Decomposition:
          Mapping from: GF(5^2) to GF(5^2)
        Composition of Mapping from: GF(5^2) to GF(5^2) given by a rule and
        Mapping from: GF(5^2) to GF(5^2) given by a rule,
          Id($),
          1
        ]
    ],
    [
        One-Cocycle
        defined by [
        Automorphism of $: Group of Lie type A3 over Finite field of size 5^2
        given by: Mapping from: $: Group of Lie type  to $: Group of Lie type
        Composition of Mapping from: $: Group of Lie type  to $: Group of
        Lie type  given by a rule and
        Mapping from: $: Group of Lie type  to $: Group of Lie type
```

```
        given by a rule
        Decomposition:
          Mapping from: GF(5^2) to GF(5^2)
        Composition of Mapping from: GF(5^2) to GF(5^2) given by a rule and
        Mapping from: GF(5^2) to GF(5^2) given by a rule,
          (1, 3),
          1
        ]
    ]
]
Time: 0.470
```

Now create the trivial cocycle:

```
> TrivialOneCocycle( AGRP );
One-Cocycle
defined by [
Automorphism of $: Group of Lie type A3 over Finite field of size 5^2
given by: Mapping from: $: Group of Lie type  to $: Group of Lie type
given by a rule
Decomposition:
  Mapping from: GF(5^2) to GF(5^2) given by a rule,
  Id($),
  1
]
>
```

And now the cocycle defining the group ${}^2A_3(5)$ and check for two elements if they are contained in ${}^2A_3(5)$:

```
> c := OneCocycle( AGRP, [GraphAutomorphism(G, Sym(3)!(1,3))] );
>
> x := Random(G);
> IsInTwistedForm( x, c );
false
>
> x := elt< G | <1,y>, <3,y @ m(Gamma.1)> > where y is Random(K);
> IsInTwistedForm( x, c );
true
>
```

109.2.3 Twisted Groups

The description of the twisted groups of Lie type is based on the extended root data, as described in the Section 103.1.7. These functions are mainly based on [Hal05].

`TwistedGroupOfLieType(c)`

Given the cocycle c on the group of automorphisms of a split group of Lie type G, return the twisted form of G, defined by that cocycle.

`TwistedGroupOfLieType(R, k, K)`

`Normalising`	BoolElt	*Default :* `true`
`Method`	MonStgElt	*Default :* "*Default*"

The twisted group of Lie type defined over the field k with coefficients in the field K corresponding to the twisted root datum R.

`TwistedGroupOfLieType(R, q, r)`

`Normalising`	BoolElt	*Default :* `true`
`Method`	MonStgElt	*Default :* "*Default*"

The twisted group of Lie type defined over the finite field of order q with coefficients in the finite field of order r (where r is a power of q) corresponding to the twisted root datum R.

`BaseRing(G)`

`CoefficientRing(G)`

The coefficient ring of the (twisted) group of Lie type G, that is the base ring of the untwisted overgroup of G.

`DefRing(G)`

The ring over which the (twisted) group of Lie type G is defined. If G is split, this is the same as the base ring of G.

`UntwistedOvergroup(G)`

The untwisted overgroup, inside which the twisted group of Lie type G was constructed.

Example H109E3

The twisted group ${}^2A_3(5)$ as a subgroup of $A_3(5^2)$.

```
> R := RootDatum("A3" : Twist := 2);
> G := TwistedGroupOfLieType(R,5,25);
> G;
G: Twisted group of Lie type 2A3,2 over GF(5) with entries over GF(5^2)
> BaseRing(G);
Finite field of size 5^2
> DefRing(G);
Finite field of size 5
> UntwistedOvergroup(G);
Group of Lie type A3 over GF(5^2)
```

`RelativeRootElement(G,delta,t)`

The relative root element corresponding to the relative root δ of the twisted group of Lie type G and the field elements given by the sequence t. This is the element $u_\delta(t)$ in [Hal05, (4.5)].

Example H109E4

Here we create the same group as in the previous example, but using a cocycle.

```
> q := 5; k := GF(q); K := GF(q^2);
>
> G := GroupOfLieType( "A3", K );
> A := AutomorphismGroup(G);
>
> AGRP := GammaGroup( k, A );
> c := OneCocycle( AGRP, [GraphAutomorphism(G, Sym(3)!(1,3))] );
>
> T := TwistedGroupOfLieType(c);
> T eq TwistedGroupOfLieType(RootDatum("A3":Twist:=2),k,K);
true
> G eq UntwistedOvergroup(T);
true
>
> x := Random(G); x in T;
false
>
> x := RelativeRootElement(T,2,[Random(K)]); x;
x1($.1^22) x3($.1^14)
> x in T;
true
```

109.3 Operations on Groups of Lie Type

Many of the basic operations for Coxeter groups are shortcuts for obtaining information about the underlying root datum (Chapter 103). Such functions are listed here; see Sections 103.3, 103.4, 103.5, and 104.4 for more details and examples of their use.

`G eq H`

Returns `true` iff the groups of Lie type G and H are equal.

`G subset H`

Returns `true` iff the group of Lie type G is a subset of H.

`IsAlgebraicallyIsomorphic(G, H)`

Returns `true` if the semisimple groups G and H are isomorphic as algebraic groups (i.e. they have the same base rings and isomorphic root data). If `true`, then the second value returned is an isomorphism.

`IsIsogenous(G, H)`

Returns `true` if G and H are isogenous. The groups must be semisimple and defined over the same field. If `true`, the subsequent values returned are: the corresponding adjoint group G_{ad}, the homomorphisms $G_{ad} \to G$ and $G_{ad} \to H$, the corresponding simply connected root datum G_{sc}, and the homomorphisms $G \to G_{sc}$ and $H \to G_{sc}$.

`IsCartanEquivalent(G, H)`

Returns `true` if, and only if, the groups of Lie type G and H are Cartan equivalent, i.e. they have isomorphic Dynkin diagrams and defined over the same ring.

`BaseRing(G)`

`CoefficientRing(G)`

The base ring k of the group of Lie type G.

`BaseExtend(G, K)`

Given a group of Lie type G with base ring k and a larger ring K, return the group $G(K)$ gotten by extending the base ring and the injection $G \to G(K)$.

`ChangeRing(G, K)`

Given a group of Lie type G and a ring K, return the group with the same root datum, but defined over a different ring.

`Generators(G)`

Generators for the group of Lie type G as an abstract group. This is currently only implemented when the base ring is a finite field.

`NumberOfGenerators(G)`

`Ngens(G)`

The number of generators for the group of Lie type G as an abstract group. This is currently only implemented when the base ring is a finite field.

`AlgebraicGenerators(G)`

A set of generators for the group of Lie type G as an algebraic group.

`NumberOfAlgebraicGenerators(G)`

`Nalggens(G)`

The number of generators for the group of Lie type G as an algebraic group.

Example H109E5

```
> k<z> := GF(4);
> G := GroupOfLieType("A2", k : Normalising:=false);
> Generators(G);
[ x1(1) , x4(1) , x1(z) , x4(z) , x2(1) , x5(1) , x2(z) , x5(z) , ( z   1) ,
(1   z)  ]
> AlgebraicGenerators(G);
[ x1(1) , x2(1) , x4(1) , x5(1) , ( z   1) , ( 1   z)  ]
```

`Order(G)`

`#G`

The order of the group of Lie type G.

`FactoredOrder(G)`

The factored order of the group of Lie type G.

`Dimension(G)`

The dimension of the group of Lie type G, considered as an algebraic variety.

Example H109E6

```
> G := GroupOfLieType("G2", 3);
> Order(G);
4245696
> FactoredOrder(G);
[ <2, 8>, <13, 1>, <3, 6>, <7, 1> ]
> G := GroupOfLieType("G2", Rationals());
> Order(G);
Infinity
> Dimension(G);
14
```

`CartanName(G)`

The Cartan name of the group of Lie type G.

`RootDatum(G)`

The root datum of the group of Lie type G.

`DynkinDiagram(G)`

Print the Dynkin diagram of the group of Lie type G.

`CoxeterDiagram(G)`

Print the Coxeter diagram of the group of Lie type G.

`CoxeterMatrix(G)`

The Coxeter matrix of the group of Lie type G.

`CoxeterGraph(G)`

The Coxeter graph of the group of Lie type G.

`CartanMatrix(G)`

The Cartan matrix of the group of Lie type G.

`DynkinDigraph(G)`

The Dynkin digraph of the group of Lie type G.

`Rank(G)`

`ReductiveRank(G)`

The reductive rank of the group of Lie type G, i.e. the dimension of the underlying root datum.

`SemisimpleRank(G)`

The semisimple rank of the group of Lie type G, i.e. the rank of the underlying root datum.

`CoxeterNumber(G)`

The Coxeter number of the group of Lie type G, i.e. the order of the Coxeter element in the Weyl group of G.

`WeylGroup(G)`

`WeylGroup(GrpPermCox, G)`

The Weyl group of the group of Lie type G as a permutation Coxeter group. This is a crystallographic Coxeter group, see Chapter 104.

`WeylGroup(GrpFPCox, G)`

The Weyl group of the group of Lie type G as a finitely presented Coxeter group. This is a crystallographic Coxeter group, see Chapter 104.

`WeylGroup(GrpMat, G)`

The Weyl group of the group of Lie type G as a reflection group. This is a crystallographic Coxeter group, see Chapter 105.

`FundamentalGroup(G)`

The fundamental group of the group of Lie type G, together with the projection of the weight lattice onto the fundamental group.

`IsogenyGroup(G)`

The isogeny group of the group of Lie type G, together with its injection into the fundamental group.

`CoisogenyGroup(G)`

The coisogeny group of the group of Lie type G, together with its projection onto the fundamental group.

109.4 Properties of Groups of Lie Type

`IsFinite(G)`

Return `true` if and only if the group of Lie type G is finite.

`IsAbelian(G)`

Returns `true` if the group of Lie type G is abelian.

`IsSimple(G)`

Returns `true` if the group of Lie type G is a simple group as an algebraic group, ie, G has no proper *connected* normal subgroups. This is true if, and only if, the underlying root datum is irreducible. Note that this does not usually mean that G is simple as an abstract group. In previous releases of Magma this function was incorrectly called `IsIrreducible`.

`IsSimplyLaced(G)`

Returns `true` if the group of Lie type G is simply laced, i.e. its Dynkin diagram contains no multiple bonds.

`IsSemisimple(G)`

Returns `true` if the group of Lie type G is semisimple.

`IsAdjoint(G)`

Returns `true` if, and only if, the group of Lie type G is adjoint (i.e. the isogeny group is trivial).

`IsWeaklyAdjoint(G)`

Returns `true` if, and only if, the group of Lie type G is weakly adjoint, i.e. its isogeny group is isomorphic to $\mathbf{Z}^n$, where n is the difference between the rank and the semisimple rank of G. Note that if G is semisimple then this function is identical to `IsAdjoint`.

`IsSimplyConnected(G)`

Returns `true` if, and only if, the group of Lie type G is simply connected (i.e. the isogeny group is equal to the fundamental group, i.e. the coisogeny group is trivial).

`IsWeaklySimplyConnected(G)`

Returns `true` if, and only if, the group of Lie type G is weakly simply connected, i.e. its coisogeny group is isomorphic to $\mathbf{Z}^n$, where n is the difference between the rank and the semisimple rank of G. Note that if G is semisimple then this function is identical to `IsSimplyConnected`.

`IsSplit(G)`

Returns `true` if and only if the group of Lie type G is split.

`IsTwisted(G)`

Returns `true` if and only if the group of Lie type G is twisted.

109.5 Constructing Elements

`elt< G | L >`

Given a group of Lie type G over the ring R and a list L of appropriate objects, construct an element of G. Suppose the underlying root datum has dimension d, rank n, and roots $\alpha_1, \ldots, \alpha_{2N}$. Each entry in the list can be one of the following:

1. A tuple $< r, t >$ where $r = 1, \ldots, 2N$ and $t \in R$. This corresponds to the unipotent term $x_r(t)$.
2. A sequence of tuples as in item (1).
3. A sequence $[t_1, \ldots, t_N]$ of elements of R. This corresponds to the unipotent element $x_1(t_1) \cdots x_N(t_N)$.
4. An integer $r = 1, \ldots, 2N$. This corresponds to the Weyl group representative n_r.
5. A Weyl group element w, either as a word or as a permutation. This corresponds to the Weyl group representative $\dot{w}$.
6. A vector $v \in R^d$ with each entry invertible. This corresponds to an element of the torus.
7. An element of G, either previously constructed or (recursively) given as a list of terms of the forms 1 to 7.

`Identity(G)`
`Id(G)`
`G ! 1`
`elt< G | >`

The identity element of the group of Lie type G.

Example H109E7

```
> G := GroupOfLieType("A5", Rationals() : Normalising := false);
> V := VectorSpace(Rationals(), 5);
> NumPosRoots(G);
15
> elt< G | <5,1/2>, 1,3,2, [1,2,3,4,5,6,7,8,9,10,11,12,13,14,15],
>          V![6,1/3,-1,3,2/3] >;
x5(1/2) n1 n3 n2 x1(1) x2(2) x3(3) x4(4) x5(5) x6(6) x7(7) x8(8) x9(9)
x10(10) x11(11) x12(12) x13(13) x14(14) x15(15) ( 6 1/3  -1   3 2/3)
```

`TorusTerm(G, r, t)`

The torus term $h_r(t) = \alpha_r^\star \otimes t$ in the group of Lie type G, where r is the index of the coroot $\alpha_r^\star$ and t an element of the base ring of G.

`CoxeterElement(G)`

The Coxeter element of the group of Lie type G, i.e. the representative of the Coxeter element in the Weyl group of G.

`Random(G)`

`Uniform` BoolElt *Default :* `true`

An element of the finite group of Lie type G chosen at random. The base ring of G must be finite. If the optional parameter `Uniform` is set to `true`, the random elements to be distributed uniformly. If the optional parameter `Uniform` is set to `false`, this function is much faster but the random elements are not distributed uniformly. Instead each double coset of the Borel subgroup occurs with equal frequency, and the elements are uniformly distributed within each double coset.

`Eltlist(g)`

The list corresponding to the element g of a group of Lie type.

`CentrePolynomials(G)`

`CenterPolynomials(G)`

A set of polynomials which are satisfied by the coordinates of a torus element h of the group of Lie type G if, and only if, h is in the centre of G.

Example H109E8

The centre of a semisimple group is finite, so the centre polynomials can be used to find all central elements.

```
> G := GroupOfLieType("B3", Rationals() : Isogeny:="SC");
> pols := CentrePolynomials(G);
> pols;
{
    -h[2] + h[3]^2,
    h[1]^2 - h[2],
    -h[1]*h[3]^2 + h[2]^2
}
> S := Scheme(AffineSpace(Rationals(), 3), Setseq(pols));
> pnts := RationalPoints(S);
> pnts;
{@ (0, 0, 0), (1, 1, -1), (1, 1, 1) @}
```

The rational points of S can be converted into elements of G, taking care to eliminate any point which has a coordinate equal to zero:

```
> V := VectorSpace(Rationals(), 3);
```

```
> [ elt< G | V!Eltseq(pnt) > : pnt in pnts | &*Eltseq(pnt) ne 0 ];
[ (1  1 -1) , 1 ]
```

109.6 Operations on Elements

109.6.1 Basic Operations

`g * h`

The product of two elements of a group of Lie type. If the Normalising flag is set for the group, then the product is normalised using the algorithms of [CMT04, CHM08]. Otherwise, the words are just concatenated.

Example H109E9

If the `Normalising` flag is set, the product is normalised, otherwise multiplication is just concatenation.

```
> G := GroupOfLieType("G2", GF(3) : Normalising:=false );
> V := VectorSpace(GF(3),2);
> g := elt< G | 1,2,1,2, V![2,2], <1,2>,<5,1> >;
> h := elt< G | <3,2>, V![1,2], 1 >;
> g*h;
n1 n2 n1 n2 (2 2) x1(2) x5(1) x3(2) (1 2) n1
> H := GroupOfLieType("G2", GF(3) : Normalising:=true  );
> g := elt< H | 1,2,1,2, V![2,2], <1,2>,<5,1> >;
> h := elt< H | <3,2>, V![1,2], 1 >;
> g*h;
x2(1) x3(1) (1 2) n1 n2 n1 n2 n1 x4(1)
```

`g ^ -1`

`Inverse(G)`

The inverse of the element g of a group of Lie type.

`g ^ n`

The nth power of the element g of a group of Lie type.

`g ^ h`

The conjugate $h^{-1}gh$, where g and h are elements of a group of Lie type.

(g, h)

Commutator(g, h)

The commutator $g^{-1}h^{-1}gh$ of g and h, where g and h are elements of a group of Lie type.

Normalise(~g)

Normalize(~g)

Normalise(g)

Normalize(g)

Normalise the element g of a group of Lie type G. The procedural form is slightly more efficient than the functional form. If the `Normalise` flag is set for G, this operation has no effect. This uses the algorithms of [CMT04, CHM08].

Example H109E10

Arithmetic in groups of Lie type.

```
> k<z> := GF(4);
> G := GroupOfLieType("C3", k);
> V := VectorSpace(k, 3);
> g := elt< G | 1,2,3, <3,z>,<4,z^2>, V![1,z^2,1] >;
> g;
n1 n2 n3 x3(z) x4(z^2) ( 1 z^2   1)
> h := elt< G | [0,1,z,1,0,z^2,1,1,z] >;
> h;
x2(1) x3(z) x4(1) x6(z^2) x7(1) x8(1) x9(z)
> g * h^-1;
x3(1) x5(z) x6(z^2) x8(1) (z^2 z^2   z) n1 n2 n3 x3(z^2) x5(z^2)
> g^3;
x3(z) x5(1) x7(z^2) x8(z^2) ( 1   1   z) n1 n2 n3 n1 n2 n3 n1 n2 n3 x1(1)
x2(z^2) x3(1) x4(z) x7(z) x9(z)
```

109.6.2 Decompositions

Bruhat(g)

Given an element g of a group of Lie type the Bruhat decomposition of g is returned. The function returns elements u, h, $\dot{w}$, u' with the properties described in Subsection 109.1.3 and so that $g = uh\dot{w}u'$.

Example H109E11

```
> k<z> := GF(4);
> G := GroupOfLieType("C3", k);
> V := VectorSpace(k, 3);
> g := elt< G | 1,2,3, <3,z>,<4,z^2>, V![1,z^2,1] >;
> Normalise(g);
x7(z^2) x8(z^2) (z^2 z^2   z) n1 n2 n3 x3(1) x6(z)
> u, h, w, up := Bruhat(g);
> u; h; w; up;
x7(z^2) x8(z^2)
(z^2 z^2   z)
n1 n2 n3
x3(1) x6(z)
```

`MultiplicativeJordanDecomposition(x)`

The multiplicative Jordan decomposition of the element x of the group of Lie type.

109.6.3 Conjugacy and Cohomology

`ConjugateIntoTorus(g)`

Given a semisimple element g in a finite group of Lie type, return a torus element t and conjugator x such that $t = xgx^{-1}$. The elements returned may be defined over a larger field that the input element.

`ConjugateIntoBorel(g)`

Given a semisimple element g in a finite group of Lie type, return a Borel element b and conjugator x such that $b = xgx^{-1}$. The elements returned may be defined over a larger field that the input element. Although any element of a group of Lie type can be conjugated into the Borel subgroup, this function is currently only implemented for semisimple elements.

`Lang(c, q)`

Given an element c in a finite group of Lie type and q a power of the characteristic, return a solution a of the Lang equation $c = a^{-F}a$. Here F is the Frobenius automorphism gotten by taking qth powers in the field.

109.7 Properties of Elements

`IsSemisimple(x)`

Return `true` if, and only if, the element x of the group of Lie type is semisimple.

`IsUnipotent(x)`

Return `true` if, and only if, the element x of the group of Lie type is unipotent.

`IsCentral(x)`

Return `true` if, and only if, the element x of the group of Lie type is in the centre of its parent group.

109.8 Roots, Coroots and Weights

The roots are stored as an indexed set

$$\{@\ \alpha_1, \ldots, \alpha_N, \alpha_{N+1}, \ldots, \alpha_{2N}\ @\},$$

where $\alpha_1, \ldots, \alpha_N$ are the positive roots in an order compatible with height; and $\alpha_{N+1}, \ldots, \alpha_{2N}$ are the corresponding negative roots (i.e. $\alpha_{i+N} = -\alpha_i$). The simple roots are $\alpha_1, \ldots, \alpha_n$ where n is the rank.

Many of these functions have an optional argument `Basis` which may take one of the following values

1. `"Standard"`: the standard basis for the (co)root space. This is the default.
2. `"Root"`: the basis of simple (co)roots.
3. `"Weight"`: the basis of fundamental (co)weights (see Subsection 105.8.3 below).

109.8.1 Accessing Roots and Coroots

`RootSpace(G)`

`CorootSpace(G)`

The lattice containing the (co)roots of the group of Lie type G.

`SimpleRoots(G)`

`SimpleCoroots(G)`

The simple (co)roots of the group of Lie type G as the rows of a matrix.

`NumberOfPositiveRoots(G)`

`NumPosRoots(G)`

The number of positive roots of the group of Lie type G.

`Roots(G)`
`Coroots(G)`

Basis MonStgElt *Default :* "*Standard*"

An indexed set containing the (co)roots of the group of Lie type G.

`PositiveRoots(G)`
`PositiveCoroots(G)`

Basis MonStgElt *Default :* "*Standard*"

An indexed set containing the positive (co)roots of the group of Lie type G.

`Root(G, r)`
`Coroot(G, r)`

Basis MonStgElt *Default :* "*Standard*"

The rth (co)root of the group of Lie type G.

`RootPosition(G, v)`
`CorootPosition(G, v)`

Basis MonStgElt *Default :* "*Standard*"

If v is a (co)root of the group of Lie type G, this returns its position; otherwise it returns 0.

Example H109E12

```
> G := GroupOfLieType("A3", 25 : Isogeny := 2);
> Roots(G);
{@
    (1 0 0),
    (0 1 0),
    (1 0 2),
    (1 1 0),
    (1 1 2),
    (2 1 2),
    (-1  0  0),
    (0 -1  0),
    (-1  0 -2),
    (-1 -1  0),
    (-1 -1 -2),
    (-2 -1 -2)
@}
> PositiveCoroots(G);
{@
    (2 -1 -1),
    (-1  2  0),
```

```
    (0 -1  1),
    (1  1 -1),
    (-1  1  1),
    (1 0 0)
@}
> #Roots(G) eq 2*NumPosRoots(G);
true
> Coroot(G, 4);
(1  1 -1)
> Coroot(G, 4 : Basis := "Root");
(1 1 0)
> CorootPosition(G, [1,1,-1]);
4
> CorootPosition(G, [1,1,0] : Basis := "Root");
4
```

HighestRoot(G)

HighestLongRoot(G)

Basis MonStgElt *Default :* "*Standard*"

The unique (long) root of greatest height in the root datum of the group of Lie type G.

HighestShortRoot(G)

Basis MonStgElt *Default :* "*Standard*"

The unique short root of greatest height in the root datum of the group of Lie type G.

Example H109E13

```
> G := GroupOfLieType("G2", RealField());
> HighestRoot(G);
(3 2)
> HighestLongRoot(G);
(3 2)
> HighestShortRoot(G);
(2 1)
```

109.8.2 Reflections

The reflections in the Weyl group have representatives in the group of Lie type.

`Reflections(G)`

The sequence of representatives of reflections in the group of Lie type G.

`Reflection(G, r)`

The representative of the reflections in the rth root in the group of Lie type G.

Example H109E14

```
> G := GroupOfLieType("A2", Rationals());
> Reflections(G);
[ n1 , n2 , n1 n2 n1  ]
```

109.8.3 Operations and Properties for Root and Coroot Indices

`RootHeight(G, r)`

`CorootHeight(G, r)`

The height of the rth (co)root of the group of Lie type G, i.e. the sum of the coefficients of α_r (resp. $\alpha_r^\star$) with respect to the simple (co)roots.

`RootNorms(G)`

`CorootNorms(G)`

The sequence of squares of the lengths of the (co)roots of the group of Lie type G.

`RootNorm(G, r)`

`CorootNorm(G, r)`

The square of the length of the rth (co)root of the group of Lie type G.

`IsLongRoot(G, r)`

Returns `true` if, and only if, the rth root of the group of Lie type G is long, i.e. the rth coroot is short.

`IsShortRoot(G, r)`

Returns `true` if, and only if, the rth root of the group of Lie type G is short, i.e. the rth coroot is long.

`AdditiveOrder(G)`

An additive order on the positive roots of the group of Lie type G, i.e. a sequence containing the numbers $1, \ldots, N$ in some order so that $\alpha_r + \alpha_s = \alpha_t$ implies t is between r and s. This is computed using the techniques of [Pap94]

Example H109E15

```
> G := GroupOfLieType("A5", GF(3));
> a := AdditiveOrder(G);
> Position(a, 2);
6
> Position(a, 3);
10
```

109.8.4 Weights

`WeightLattice(G)`

`CoweightLattice(G)`

The (co)weight lattice of the group of Lie type G.

`FundamentalWeights(G)`

`FundamentalCoweights(G)`

`Basis` MonStgElt *Default :* "*Standard*"

The fundamental (co)weights of the group of Lie type G as the rows of a matrix.

`DominantWeight(G, v)`

`Basis` MonStgElt *Default :* "*Standard*"

The unique dominant weight in the same W-orbit as v, where W is the Weyl group of G and v is a weight given as a vector or a sequence representing a vector. The second value returned is a Weyl group element taking v to the dominant weight.

109.9 Building Groups of Lie Type

Currently the only subgroups of a group of Lie type that can be constructed are subsystem subgroups.

`SubsystemSubgroup(G, a)`

The subsystem subgroup of the group of Lie type G generated by the standard maximal torus and the root subgroups with roots $\alpha_{a_1}, \ldots, \alpha_{a_k}$ where $a = \{a_1, \ldots, a_k\}$ is a set of integers.

`SubsystemSubgroup(G, s)`

The subsystem subgroup of the group of Lie type G generated by the standard maximal torus and the root subgroups with roots $\alpha_{s_1}, \ldots, \alpha_{s_k}$ where $s = [s_1, \ldots, s_k]$ is a *sequence* of integers. In this version the roots must be simple in the root subdatum (i.e. none of them may be a summand of another) otherwise an error is signalled. The simple roots will appear in the subdatum in the given order.

Example H109E16

```
> G := GroupOfLieType("A4",Rationals());
> PositiveRoots(G);
{@
    (1 0 0 0),
    (0 1 0 0),
    (0 0 1 0),
    (0 0 0 1),
    (1 1 0 0),
    (0 1 1 0),
    (0 0 1 1),
    (1 1 1 0),
    (0 1 1 1),
    (1 1 1 1)
@}
> H := SubsystemSubgroup(G, [6,1,4]);
> H;
H: Group of Lie type A3 over Rational Field
> PositiveRoots(H);
{@
    (0 1 1 0),
    (1 0 0 0),
    (0 0 0 1),
    (1 1 1 0),
    (0 1 1 1),
    (1 1 1 1)
@}
> h := elt<H|<2,2>,1>;
> h;  G!h;
x2(2) n1
x1(2) ( 1 -1  1 -1) n2 n3 n2
```

`DirectProduct(G1, G2)`

The direct product of the groups G_1 and G_2. The two groups must have the same base ring.

`Dual(G)`

The dual of the group of Lie type G, obtained by swapping the roots and coroots.

`SolubleRadical(G)`

The soluble radical of the group of Lie type G.

`StandardMaximalTorus(G)`

The standard maximal torus of the group of Lie type G.

Example H109E17

```
> G1 := GroupOfLieType( "A5", GF(7) );
> G2 := GroupOfLieType( "B4", GF(7) );
> DirectProduct(G1, Dual(G2));
$: Group of Lie type A5 C4 over Finite field of size 7
>
> G := GroupOfLieType(StandardRootDatum("A",3), GF(17));
> SolubleRadical(G);
$: Torus group of Dimension 1 over Finite field of size 17
```

109.10 Automorphisms

The following functions construct the standard automorphisms of a group of Lie type, as described in [Car72] (except for the graph automorphism of G_2). In many cases, including the finite groups, every automorphism is a product of these standard automorphisms.

109.10.1 Basic Functionality

`AutomorphismGroup(G)`

Automorphism group of a group of Lie type G.

`IdentityAutomorphism(G)`

`One(A)`

`Id(A)`

The identity automorphism of the group of Lie type G.

`Mapping(a)`

The map object associated with the automorphism a.

`Automorphism(m)`

Given a map object m from G to G, which is an isomorphism, returns the associated automorphism as an automorphism of a group of Lie type.

`h * g`

The composition of the group of Lie type automorphisms h and g.

`h ^ n`

The nth power of the group of Lie type automorphism h.

`g ^ h`

The conjugate $h^{-1}gh$, where g and h are group of Lie type automorphisms g and h

`Domain(A)`
`Codomain(A)`
`Domain(h)`
`Codomain(h)`

Domain or codomain of an automorphism of a group of Lie type or of the group of automorphisms.

109.10.2 Constructing Special Automorphisms

`InnerAutomorphism(G, x)`

The inner automorphism taking $g \in G$ to g^x, where x is an element of the group of Lie type G.

`DiagonalAutomorphism(G, v)`

The diagonal automorphism of the semisimple group of Lie type G given by the vector v. Let n be the semisimple rank of G and let k be its base field. Then v must be a vector in k^n with every component nonzero. The function returns the automorphism given by the character χ defined by $\chi(\alpha_i) = v_i$, where α_i is the ith simple root. Since our groups are algebraic, a diagonal automorphism is just a special case of an inner automorphism.

`GraphAutomorphism(G, p)`
`DiagramAutomorphism(G, p)`

`SimpleSigns` ANY *Default :* 1

The graph automorphism of the group of Lie type G given by the permutation p. The permutation must act on the indices of simple roots of G or the indices of all roots of G. The graph automorphism of the group of type G_2 has not been implemented yet.

The optional parameter `SimpleSigns` can be used to specify the signs corresponding to each simple root. This should either be a sequence of integers ± 1, or a single integer ± 1.

`FieldAutomorphism(G, sigma)`

The field automorphism of the group of Lie type G induced by σ, an element of the automorphism group of the base field of G

`RandomAutomorphism(G)`
`Random(A)`

A random element in A, the automorphism group of the group of Lie type G.

`DualityAutomorphism(G)`

The duality automorphism of G. This is an automorphism that takes every unipotent term $x_r(t)$ to $x_s(\pm t)$, where $s =$ `Negative(RootDatum(G),r))`.

`FrobeniusMap(G,q)`

The Frobenius automorphism of the finite group of Lie type G gotten by qth powers in the base field. The integer q must be a power of the characteristic of the base field of G.

109.10.3 Operations and Properties of Automorphisms

`DecomposeAutomorphism(h)`

Given a group of Lie type automorphism h, this returns a field automorphism f, a graph automorphism g and an inner automorphism i such that $h = fgi$. This only works for groups defined over finite fields. The algorithm is due to Scott Murray and Sergei Haller.

`IsAlgebraic(h)`

Returns `true` if and only if the automorphism h is algebraic.

Example H109E18

Some automorphisms of $B_2(4)$

```
> G := GroupOfLieType("B2", GF(4));
> A := AutomorphismGroup(G);
> A!1 eq IdentityAutomorphism(G);
true
> g := GraphAutomorphism(G, Sym(2)!(1,2));
> g;
Automorphism of Group of Lie type B2 over Finite field of size 2^2
given by: Mapping from: Group of Lie type  to Group of Lie type
given by a rule
Decomposition:
  Mapping from: GF(2^2) to GF(2^2) given by a rule,
  (1, 2),
  1
```

The automorphism of $B_2(4)$ whose stabiliser is ${}^2B_2(4)$ is constructed by the following code.

```
> sigma := iso< GF(4) -> GF(4) | x :-> x^2, x :-> x^2 >;
> h := FieldAutomorphism(G, sigma) * g;
> h in A;
true
> f,g,i := DecomposeAutomorphism(h);
> assert f*g*i eq h;
```

109.11 Algebraic Homomorphisms

`GroupOfLieTypeHomomorphism(phi, k)`

The algebraic homomorphism of groups of Lie type over the ring k corresponding to the root datum morphism ϕ. See Chapter 109.

Example H109E19

This example constructs the algebraic projection $GL_4(\mathbf{Q}) \to PGL_4(\mathbf{Q})$.

```
> RGL := StandardRootDatum( "A", 3 );
> RPGL := RootDatum( "A3" );
> A := VerticalJoin( SimpleRoots(RGL), Vector([Rationals()|1,1,1,1]) )^-1 *
>   VerticalJoin( SimpleRoots(RPGL), Vector([Rationals()|0,0,0]) );
> B := VerticalJoin( SimpleCoroots(RGL), Vector([Rationals()|1,1,1,1]) )^-1 *
>   VerticalJoin( SimpleCoroots(RPGL), Vector([Rationals()|0,0,0]) );
> phi := GroupOfLieTypeHomomorphism( hom< RGL -> RPGL | A, B >, Rationals() );
> GL := Domain( phi );
> phi( elt<GL|<1,2>, Vector([Rationals()| 7,1,11,1])> );
x1(2) (   7 1/11   11)
```

109.12 Twisted Tori

The functionality presented here deals with the computation of the twisted tori of a finite group of Lie type.

Note that for a given group $G(k)$, the twisted tori are returned as subgroups of the standard torus of $G(K)$ for the smallest field extension K of k, where this is possible.

For finite fields and an untwisted group of Lie type $G(k)$, a twisted torus $T_w(k)$ of $G(k)$ has the form

$$T_w(k) = \{t \in T(K) | t^{\sigma w} = t\},$$

where $T(K)$ is the standard K-split torus of $G(K)$, σ is the generator of the Galois group $\mathrm{Gal}(K : k)$ and w is an element of the Weyl group of $G(k)$.

`TwistedTorusOrder(R, w)`

Given the root datum R and a Weyl group element w, computes the orders of the cyclic components of the twisted torus $T_w(k) \subset G(R, k)$ as sequence of polynomials in q, the order of the field k.

`TwistedToriOrders(G)`

`TwistedToriOrders(R)`

Given a group of Lie type G or a root datum R, takes for every conjugacy class of the Weyl group of G a representative w, and computes `TwistedTorusOrder(R, w)`. Returns the sequence of the lists consisting of `TwistedTorusOrder(R, w)` and w for every conjugacy class.

`TwistedTorus(G, w)`

Computes the twisted torus $T_w(k)$ of the group of Lie type G for the given element w of the Weyl group of G. Returned is the list consisting of three elements, first of them being the sequence of orders of cyclic parts of the torus, the second being the sequence of generators of the respective orders and the third being w. See [Hal05] for the algorithm used.

`TwistedTori(G)`

Computes one twisted torus $T_w(k)$ of the group of Lie type G for each conjugacy class w^W of the Weyl group W of G. A sequence of them is returned. See [Hal05] for the algorithm used.

Example H109E20

We compute all twisted tori of $A_1(5)$:

```
> G := GroupOfLieType("A1", 5);
> TwistedToriOrders(G);
[ [*
    [
        q - 1
    ],
    Id($)
*], [*
    [
        q + 1
    ],
    (1, 2)
*] ]
> TwistedTori(G);
[ [*
    [ 4 ],
    [ (2)  ],
    Id($)
*], [*
    [ 6 ],
    [ ( k.1^4)  ],
    (1, 2)
*] ]
```

As we may notice, the second one is contained in the group over the quadratic field extension:

```
> Universe($1[2][2]);
$: Group of Lie type A1 over Finite field of size 5^2
```

Example H109E21

These are the orders of the decompositions of all (up to conjugacy) maximal tori of the group $G_2(q)$ as polynomials in q:

```
> R := RootDatum("G2");
> [ t[1] : t in TwistedToriOrders(R) ];
[
    [  q - 1,  q - 1  ],
    [  q + 1,  q + 1  ],
    [  q^2 - 1        ],
    [  q^2 - 1        ],
    [  q^2 + q + 1    ],
    [  q^2 - q + 1    ]
]
```

109.13 Sylow Subgroups

We present here the functionality which allows to compute the Sylow subgroups of finite groups of Lie type.

`PrintSylowSubgroupStructure(G)`

This procedure prints out a list of all primes p dividing the order of the group of Lie type G along with the "goodness" of p, the exponent of p in the factorisation of $|G|$ and a sequence of integers. The positive integers give the orders of the decomposition of a torus T_w into cyclic groups such that the Sylow subgroup is contained in $\langle T_w, C_W(w)\rangle$. The negative number indicates the p-part coming from $C_W(w)$. If more than one such torus exists, then one line is printed for each of them.

A prime is said to be "GOOD" if it is equal to the characteristic of the base field k of G, "good" if the Sylow subgroup is abelian, thus contained in a torus, and "bad" if it is not abelian and thus not contained in a torus. See [Hal05] for the algorithm used.

`SylowSubgroup(G, p)`

Compute a p-Sylow subgroup S of the group of Lie type G. Returned is a list of a two sequences. The second sequence contains generators of S. The first one is a sequence of integers giving the orders of the respective generator if the generator is a torus element and the negative of the order of $\langle g\rangle/(\langle g\rangle \cap T_w)$ in case the generator g is not a torus element. See [Hal05] for the algorithm used.

Example H109E22

Compute

```
> G := GroupOfLieType("G2", 5);
> PrintSylowSubgroupStructure(G);
G: Group of Lie type G2 over Finite field of size 5
Order(G) is  2^6 * 3^3 * 5^6 * 7^1 * 31^1
Order(W) is  2^2 * 3^1
...compute tori...
...compute sylows...
   2 (bad)  : 6 [ 4, 4, -4 ]
   3 (bad)  : 3 [ 6, 6, -3 ]
   5 (GOOD) : The unipotent subgroup of G
   7 (good) : 1 [ 21 ]
   31 (good) : 1 [ 31 ]
> SylowSubgroup(G,2);
[*
    [ 4, 4, -2, -2 ],
    [ (2 1) , (1 2) , n2 , n1 n2 n1 n2 n1 n2  ]
*]
```

note that the orders of the non-toral elements is not necessarily the corresponding integer in the first sequence:

```
> gens := $1[2];
> [ Order(g) : g in gens ];
[ 4, 4, 4, 4 ]
```

but, in this example, their squares are contained in the torus:

```
> gens[3]^2 eq gens[2]^2, gens[4]^2 eq gens[2]^2;
true true
```

109.14 Representations

This section describes basic functionality for Lie algebra representations: see Chapter 110 for more functions for highest weight representations and decompositions.

`StandardRepresentation(G)`

The standard (projective) representation of the semisimple group of Lie type G over an extension its base ring. In other words, the smallest dimension highest-weight representation. For the classical groups, this is the natural representation. If this is a projective representation rather than a linear representation, a warning is given. This is constructed from the corresponding Lie algebra representation, using the algorithm in [CMT04].

`AdjointRepresentation(G)`

The adjoint (projective) representation of the group of Lie type G over an extension of its base ring, i.e. the representation given by the action of G on its Lie algebra. The Lie algebra itself is the second returned value. This is constructed from the corresponding Lie algebra representation, using the algorithm in [CMT04].

`LieAlgebra(G)`

The Lie algebra of the group of Lie type G, together with the adjoint representation. If this is a projective representation rather than a linear representation, a warning is given.

`HighestWeightRepresentation(G, v)`

The highest weight (projective) representation with highest weight v of the group of Lie type G over an extension of its base ring. If this is a projective representation rather than a linear representation, a warning is given. This is constructed from the corresponding Lie algebra representation, using the algorithm in [CMT04].

Example H109E23

```
> G := GroupOfLieType("A2", Rationals() : Isogeny := "SC");
> rho := StandardRepresentation(G);
> rho(elt< G | 1 >);
[ 0 -1  0]
[ 1  0  0]
[ 0  0  1]
> rho(elt<G | <2,1/2> >);
[  1   0   0]
[  0   1   0]
[  0 1/2   1]
> rho(elt< G | VectorSpace(Rationals(),2)![3,5] >);
[  3   0   0]
[  0 5/3   0]
[  0   0 1/5]
>
> G := GroupOfLieType("A2", Rationals());
> Invariants(CoisogenyGroup(G));
[ 3 ]
> rho := StandardRepresentation(G);
Warning: Projective representation
> BaseRing(Codomain(rho));
Algebraically closed field with no variables
> rho(elt< G | VectorSpace(Rationals(),2)![3,1] >);
[r1  0  0]
[ 0 r2  0]
[ 0  0 r2]
> rho(elt< G | VectorSpace(Rationals(),2)![3,1] >)^3;
```

```
[  9   0   0]
[  0 1/3   0]
[  0   0 1/3]
```

`GeneralisedRowReduction(`ρ`)`

`RowReductionHomomorphism(`ρ`)`

`Inverse(`ρ`)`

Given a projective matrix representation $\rho : G \to \mathrm{GL}_m(k)$, return its inverse.

109.15 Bibliography

[**Car72**] Roger W. Carter. *Simple groups of Lie type*. John Wiley & Sons, London-New York-Sydney, 1972. Pure and Applied Mathematics, Vol. 28.

[**Car93**] Roger W. Carter. *Finite groups of Lie type*. John Wiley & Sons, Chichester, 1993. Conjugacy classes and complex characters, Reprint of the 1985 original, A Wiley-Interscience Publication.

[**CHM08**] Arjeh M. Cohen, Sergei Haller, and Scott H. Murray. Computing in unipotent and reductive algebraic groups. *LMS J. Comput. Math.*, 11:343–366, 2008.

[**CMT04**] Arjeh M. Cohen, Scott H. Murray, and D. E. Taylor. Computing in groups of Lie type. *Math. Comp.*, 73(247):1477–1498, 2004.

[**Hal05**] Sergei Haller. *Computing Galois Cohomology and Forms of Linear Algebraic Groups*. Phd thesis, Technical University of Eindhoven, 2005.

[**Pap94**] Paolo Papi. A characterization of a special ordering in a root system. *Proc. Amer. Math. Soc.*, 120(3):661–665, 1994.

[**Ste62**] Robert Steinberg. Générateurs, relations et revˆetements de groupes algébriques. In *Colloq. Théorie des Groupes Algébriques (Bruxelles, 1962)*, pages 113–127. Librairie Universitaire, Louvain, 1962.

110 REPRESENTATIONS OF LIE GROUPS AND ALGEBRAS

110.1 Introduction 3375

110.1.1 Highest Weight Modules 3375

110.1.2 Toral Elements 3376

110.1.3 Other Highest Weight Representations 3376

110.2 Constructing Weight Multisets 3377

TrivialLieRepresentation Decomposition(R) 3377
LieRepresentationDecomposition(R) 3377
LieRepresentationDecomposition(R, v) 3377
LieRepresentation Decomposition(R, Wt, Mp) 3377
AdjointRepresentationDecomposition(R) 3377

110.3 Constructing Representations 3378

110.3.1 Lie Algebras 3378

TrivialRepresentation(L) 3378
AdjointRepresentation(L) 3378
StandardRepresentation(L) 3378
HighestWeightRepresentation(L, w) 3379
HighestWeightModule(L, w) 3380
TensorProduct(Q) 3380
SymmetricPower(V, n) 3380
ExteriorPower(V, n) 3380

110.3.2 Groups of Lie Type 3382

TrivialRepresentation(G) 3382
StandardRepresentation(G) 3382
AdjointRepresentation(G) 3383
LieAlgebra(G) 3383
HighestWeightRepresentation(G, v) 3383

110.4 Operations on Weight Multisets 3384

110.4.1 Basic Operations 3384

RootDatum(D) 3384
Weights(D) 3384
WeightsAndMultiplicities(D) 3384
Multiset(D) 3384
Multiplicity(D, v) 3384
eq 3384
+ 3384
+:= 3384
AddRepresentation(∼D, E, c) 3385
AddRepresentation(∼D, E) 3385
+ 3385
AddRepresentation(∼D, v, c) 3385
AddRepresentation(∼D, v) 3385
+:= 3385
* 3385
/ 3385
*:= 3385
/:= 3385
* 3385
ProductRepresentation(D, E) 3385
ProductRepresentation(D, E, R) 3386
SubWeights(D, Q, S) 3386
PermuteWeights(D, pi, S) 3386

110.4.2 Conversion Functions 3387

VirtualDecomposition(C) 3387
VirtualDecomposition(R, v) 3387
DecomposeCharacter(C) 3387
DominantCharacter(D) 3388

110.4.3 Calculating with Representations 3388

RepresentationDimension(D) 3388
RepresentationDimension(R, v) 3388
CasimirValue(R, w) 3388
QuantumDimension(R, w) 3388
Branch(FromGrp, ToGrp, v, M) 3389
Branch(ToGrp, D, M) 3389
Collect(R, D, M) 3389
TensorProduct(R, v, w) 3390
TensorProduct(D, E) 3390
TensorProduct(Q) 3390
TensorPower(R, n, v) 3390
TensorPower(D, n) 3390
AdamsOperator(R, n, v) 3391
AdamsOperator(D, n) 3391
SymmetricPower(R, n, v) 3391
SymmetricPower(D, n) 3391
AlternatingPower(R, n, v) 3391
AlternatingPower(D, n) 3391
Plethysm(R, lambda, v) 3392
Plethysm(D, lambda) 3392
Spectrum(R, v, t) 3392
Spectrum(D, t) 3392
Demazure(R, v, w) 3393
Demazure(D, w) 3393
Demazure(R, v) 3393
Demazure(D) 3393
LittlewoodRichardsonTensor(p, q) 3394
LittlewoodRichardson Tensor(P, M, Q, N) 3394
LittlewoodRichardsonTensor(R, v, w) 3394
LittlewoodRichardsonTensor(D, E) 3394
AlternatingDominant(D, w) 3396
AlternatingDominant(R, wt, w) 3396
AlternatingDominant(D) 3397
AlternatingDominant(R, wt) 3397

`AlternatingWeylSum(R, v)` 3398
`AlternatingWeylSum(D)` 3398

110.5 Operations on Representations 3398

110.5.1 Lie Algebras 3398

`CharacterMultiset(V)` 3398
`CharacterMultiset(ρ)` 3398
`Weights(V)` 3398
`WeightsAndVectors(V)` 3398
`Weights(ρ)` 3398
`WeightsAndVectors(ρ)` 3398
`DecompositionMultiset(V)` 3398
`DecompositionMultiset(ρ)` 3398
`HighestWeightsAndVectors(V)` 3399
`DirectSum(U, V)` 3399
`DirectSumDecomposition(V)` 3399
`IndecomposableSummands(V)` 3399
`DirectSum(ρ, τ)` 3399
`DirectSumDecomposition(ρ)` 3399
`IndecomposableSummands(ρ)` 3399
`TensorProduct(Q)` 3399
`SymmetricPower(V, n)` 3399
`ExteriorPower(V, n)` 3400

110.5.2 Groups of Lie Type 3402

`DirectSum(ρ, τ)` 3402
`DirectSumDecomposition(ρ)` 3402
`IndecomposableSummands(ρ)` 3402
`CharacterMultiset(V)` 3402
`CharacterMultiset(ρ)` 3402
`Weights(ρ)` 3402
`WeightsAndVectors(ρ)` 3402
`WeightVectors(ρ)` 3402
`Weight(ρ, v)` 3402
`DecompositionMultiset(V)` 3402
`DecompositionMultiset(ρ)` 3402
`HighestWeights(ρ)` 3402
`HighestWeightVectors(ρ)` 3402
`GeneralisedRowReduction(ρ)` 3402

110.6 Other Functions for Representation Decompositions . . . 3403

`FundamentalClosure(R, S)` 3403
`Closure(R, S)` 3403
`RestrictionMatrix(R, Q)` 3403
`RestrictionMatrix(R, S)` 3403
`KLPolynomial(x, y)` 3404
`RPolynomial(x, y)` 3404
`Exponents(R)` 3406
`ToLiE(D)` 3406
`FromLiE(R, p)` 3406

110.6.1 Operations Related to the Symmetric Group 3407

`ConjugationClassLength(l)` 3407
`PartitionToWeight(l)` 3407
`WeightToPartition(v)` 3407
`TransposePartition(l)` 3407

110.6.2 FusionRules 3408

`WZWFusion(R, v, w, k)` 3408
`WZWFusion(D, E, k)` 3408

110.7 Subgroups of Small Rank . . 3409

`LiEMaximalSubgroups()` 3409
`MaximalSubgroups(G)` 3409
`RestrictionMatrix(G, H)` 3409

110.8 Subalgebras of su(d) 3410

`IrreducibleSimpleSubalgebrasOfSU(N)` 3410
`IrreducibleSimple SubalgebraTreeSU(Q, d)` 3410
`PrintTreesSU(Q, F)` 3410

110.9 Bibliography 3412

Chapter 110
REPRESENTATIONS OF LIE GROUPS AND ALGEBRAS

110.1 Introduction

This chapter gives functionality for direct sums of *highest weight representations* (or modules). This is an important class of representations of (almost) semisimple Lie algebras (Chapter 106) and connected reductive algebraic groups (Chapter 109). This class includes all finite dimensional representations if the base field is the complex field.

The representations we are considering are in bijection with sets of dominant weights with multiplicities. Such sets are called *decomposition multisets*. Many interesting computations in representation theory can be done combinatorially with weight multisets, without the need to construct the module itself. Examples of the things we can compute include: module dimension, the multiset of all the weights, and decomposition multisets for symmetric powers, alternating powers, and tensor products. We can also restrict a decomposition multiset to a subgroup or induce it to a supergroup.

The code for such combinatorial computations is based on the LiE software package [vLCL92]. The algorithms for computing the actual representations are from [dG01] in the Lie algebra case, and from [CMT04] in the group case.

110.1.1 Highest Weight Modules

This introduction is inspired by the LiE manual [vLCL92].

First consider connected reductive Lie groups over the complex field. If G is a connected reductive complex Lie group, then it is a homomorphic image $G = \xi(G')$, where ξ is a Lie-group homomorphism with finite kernel and G' is the direct product of a simply connected group and a torus. Recall that a simply connected group is a direct product of *simple* simply connected groups. In particular, such groups are determined by their Cartan name and the dimension of the torus. For example, we denote the direct product of the group of type $A_4C_3B_2$ with a two dimensional torus by $A_4C_3B_2T_2$. Most of the code ported from LiE works only for groups of this form. Similar terminology is used for the root datum corresponding to a group.

Connected reductive complex Lie groups have a very pleasing representation theory:

– Every module decomposes as a direct sum of irreducible representations.

– The (finite dimensional) irreducible representations correspond to dominant weights.

It follows that representations correspond to finite sets of dominant weights with multiplicity. These multisets are called *decomposition multisets*. We can use this classification to do useful computations about representations, without having to explicitly construct them.

Multisets of weights can be used for other purposes as well: The multiset of all weights occurring in a module M is called the *character multiset*. Since the Weyl group permutes the weights occurring in the character of M, it suffices to consider only the dominant weights with their multiplicities. This is called the *dominant character multiset*. In the LiE system, multisets of weights are represented by polynomials: for example, the decomposition multiset is called a decomposition character.

When using the functions in this section, it is important to keep track of which kind of multiset you are using. For example, if you input a decomposition multiset to a function that expects a dominant character multiset, the output is meaningless.

We often abbreviate decomposition multiset to *decomposition*, and similarly for character multisets. Write R_D for the root datum of the group of the decomposition D. Denote the irreducible module for the group with root datum R with highest weight v by V_v^R, or to V_v if R is clear from the context.

It is often useful to define consider *virtual* multisets, which allow weights to have negative multiplicities. We call a virtual multiset *proper* if its weights all have nonnegative multiplicities. A decomposition corresponds to an actual module if and only if it is proper.

110.1.2 Toral Elements

Many functions use a special syntax for finite-order elements of the torus of a Lie group G (we are rarely interested in infinite-order elements). Recall that a weight is in fact a mapping from the torus T to $C^\star$, and thus a weight λ can be evaluated at an element $t \in T$. The resulting element is written t^λ. A set of fundamental weights $\omega_1, \ldots, \omega_r$ has the property that any element $t \in T$ is uniquely determined by the values $t^{\omega_1}, \ldots, t^{\omega_r}$. Therefore, we may represent t as a vector $(a_1, \ldots, a_r, n)$, with the property that $t^{\omega_i} = e^{2\pi i a_i/n} = \zeta_n^{a_i}$, where $\zeta_n = e^{2\pi i/n}$ is the canonical n-th root of unity. An example of a function which uses this syntax for toral elements is `Spectrum`. This function also provides a means to convert toral elements into a more natural form: see Example H110E10.

110.1.3 Other Highest Weight Representations

MAGMA can also construct highest weight representations for:

– (Almost) reductive Lie algebras (Chapter 106); and

– Split groups of Lie type (Chapter 109).

If the base field ha positive characteristic, highest weight representations are indecomposable, but not necessarily irreducible. In some cases there are irreducible representations which are not highest weight representations.

For groups of Lie type, we consider projective representations (i.e., homomorphisms to a projective general linear group). Suppose G is a split group of Lie type defined over the field k and r is the least common multiple of the nonzero abelian-group invariants of the coisogeny group of G (see Section 103.1.6). Let K be an extension of k containing at least one rth root of each element of k (i.e., K contains a Kummer extension). Then highest weight representations are projective representations defined over K, and are constructed using polynomial functions and rth roots.

If k already contains all rth roots, then no extension is needed and the representation will be linear rather than projective. This happens when $r = 1$, i.e., the coisogeny group is torsion free. This includes direct products of a simply connected group and a torus. The general linear group also has this property. It also happens when k is the complex field or field of algebraic numbers, when k is the real field and r is odd, and when k is finite and $|k| - 1$ is coprime to r.

The functions give a warning when the representation is not linear, but this can be avoided using the optional parameter `NoWarning`. Note that an appropriate extension K can be constructed for all fields other than rational function fields, fields of Laurent series, and local fields. In these cases, as well as for nonfields, the representations can only be computed when $r = 1$.

110.2 Constructing Weight Multisets

In this section, we describe how to construct weight multisets.

`TrivialLieRepresentationDecomposition(R)`

`LieRepresentationDecomposition(R)`

The decomposition multiset of the trivial representation. The root datum R must be weakly simply connected.

`LieRepresentationDecomposition(R, v)`

The decomposition multiset of the highest weight representation with weight v, i.e., the singleton multiset. The root datum R must be weakly simply connected. The weight v must be a sequence of length d or an element of $\mathbf{Z}^d$, where d is the dimension of the root datum R.

`LieRepresentationDecomposition(R, Wt, Mp)`

The decomposition multiset with weights given by the sequence Wt and multiplicities given by of the sequence Mp. The root datum R must be weakly simply connected. The weights must be a sequences of length d or elements of $\mathbf{Z}^d$, where d is the dimension of the root datum R.

`AdjointRepresentationDecomposition(R)`

The decomposition multiset of the adjoint representation. This has the highest root of R as its highest weight with multiplicity one. The root datum R must be weakly simply connected.

Example H110E1

The adjoint representation:

```
> R := RootDatum("D4" : Isogeny := "SC");
> D := AdjointRepresentationDecomposition(R);
> D:Maximal;
Highest weight decomposition of representation of:
     R: Simply connected root datum of dimension 4 of type D4
     Dimension of weight space:4
     Weights:
          [
               (0 1 0 0)
          ]
     Multiplicities:
          [ 1 ]
> HighestRoot(R : Basis := "Weight");
(0 1 0 0)
```

110.3 Constructing Representations

110.3.1 Lie Algebras

The functions described in this section are applicable only to almost reductive structure constant Lie algebras.

If L has of large dimension, the step that calculates the information needed to compute preimages for these representations can be quite time consuming. So if there is no requirement for preimages, this step may be skipped by setting the optional argument `ComputePreImage` to `false`.

`TrivialRepresentation(L)`

The one-dimensional trivial representation of the Lie algebra L over its base ring.

`AdjointRepresentation(L)`

`ComputePreImage` BoolElt *Default :* `true`

The adjoint representation of the Lie algebra L acting on itself.

`StandardRepresentation(L)`

`ComputePreImage` BoolElt *Default :* `true`

The standard representation of the semisimple Lie algebra L over its base ring. This is the smallest dimensional faithful representation of G (with a few small exceptions). The Killing form of L must be nondegenerate.

Example H110E2

```
> R := RootDatum("A2");
> #CoisogenyGroup(R);
3
> L := LieAlgebra(R, GF(2));
> h := StandardRepresentation(L);
> h(L.1);
[0 0 1]
[0 0 0]
[0 0 0]
> L := LieAlgebra(R, GF(3));
> h := StandardRepresentation(L);
>> h := StandardRepresentation(L);
                               ^
Runtime error in 'StandardRepresentation': Cannot compute the standard
representation in characteristic  3
```

The coisogeny group of a simply connected root datum always has order one, so we can compute the standard representation in this case.

```
> R := RootDatum("A2" : Isogeny:="SC");
> L := LieAlgebra(R, GF(3));
> h := StandardRepresentation(L);
```

`HighestWeightRepresentation(L, w)`

The representation of the Lie algebra L with highest weight w (given either as a vector or as a sequence representing a vector). The result is a function, which for an element of L gives the corresponding matrix. The algorithm used is described in [dG01].

Example H110E3

```
> L:= LieAlgebra("G2", RationalField());
> DimensionOfHighestWeightModule(RootDatum(L), [1,0]);
7
> rho:= HighestWeightRepresentation(L, [1,0]);
> e, f, h := ChevalleyBasis(L);
> rho(e[1]+f[1]);
[ 0  1  0  0  0  0  0]
[ 1  0  0  0  0  0  0]
[ 0  0  0 -2  0  0  0]
[ 0  0 -1  0 -1  0  0]
[ 0  0  0 -2  0  0  0]
[ 0  0  0  0  0  0  1]
[ 0  0  0  0  0  1  0]
```

```
> Codomain(rho);
Full Matrix Lie Algebra of degree 7 over Rational Field
> N := sub<Codomain(rho) | [ rho(x) : x in e ]>;
> Dimension(N);
6
> IsSolvable(N);
true
```

`HighestWeightModule(L, w)`

Given a semisimple Lie algebra L corresponding to a root datum of rank r and a sequence w of non-negative integers of length r, this returns the irreducible L-module with highest weight w. The object returned is a left module over L. The algorithm used is described in [dG01].

`TensorProduct(Q)`

Given a sequence Q of left-modules over a Lie algebra, this function returns the module M that is the tensor product of the elements of Q. It also returns a map ϕ from the Cartesian product P of the modules in Q to M as the second return value. If t is a tuple whose i-th component is an element from the i-th module in Q then ϕ maps t to the element of M that corresponds to the tensor product of the elements of t.

`SymmetricPower(V, n)`

Given a left-module V over a Lie algebra, and an integer $n \geq 2$, this function returns the module M that is the n-th symmetric power of V. It also returns a map f from the n-fold Cartesian product of V to M. This map is multilinear and symmetric, i.e., if two of its arguments are interchanged then the image remains the same. Furthermore, f has the universal property, i.e., any multilinear symmetric map from the n-fold Cartesian product into a vector space W can be written as the composition of f with a map from M into W.

`ExteriorPower(V, n)`

Given a left-module V over a Lie algebra, and an integer $2 \leq n \leq \dim(V)$, this function returns the module M that is the n-th exterior power of V. It also returns a map f from the n-fold Cartesian product of V to M. This map is multilinear and antisymmetric, i.e., if two of its arguments are interchanged then the image is multiplied by -1. Furthermore, f has the universal property, i.e., any multilinear antisymmetric map from the n-fold Cartesian product into a vector space W can be written as the composition of f with a map from M into W.

Example H110E4

```
> L:= LieAlgebra("G2", Rationals());
> V1:= HighestWeightModule(L, [1,0]);
> V2:= HighestWeightModule(L, [0,1]);
> T,f:= TensorProduct([V1,V2]);
> HighestWeightsAndVectors(T);
[
    (1 1),
    (2 0),
    (1 0)
]
[
    [
        T: (1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
            0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
            0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0)
    ],
    [
        T: (0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 3 0 0 0 0 0 0 0 0 0 0 0 0 -3 0 0 0 0 0
            0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
            0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0)
    ],
    [
        T: (0 0 0 0 0 0 1 2 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 -1 0 0
            0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 -3 0 0 0 0 0 0 0 0 0 0
            0 0 -3 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0)
    ]
]
> DecomposeTensorProduct(RootDatum(L), [1,0], [0,1]);
[
    (1 1),
    (2 0),
    (1 0)
]
[ 1, 1, 1 ]
> f(<V1.2,V2.3>);
T: (0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
    0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
    0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0)
```

So we see that the tensor product T decomposes as a direct sum of three submodules. This information can also be computed by using `DecomposeTensorProduct`. However, in the former case, the corresponding highest-weight vectors are also given.

```
> E,h:= ExteriorPower(V1, 3);
> h(<V1.1,V1.3,V1.4>);
E: (0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0)
> h(<V1.1,V1.4,V1.3>);
```

```
E: (0 0 0 0 0 -1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0)
> DecomposeExteriorPower( RootDatum(L), 3, [1,0] );
[
    (2 0),
    (1 0),
    (0 0)
]
[ 1, 1, 1 ]
> HighestWeightsAndVectors(E);
[
    (2 0),
    (1 0),
    (0 0)
]
[
    [
        E: (1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
        0)
    ],
    [
        E: (0 0 0 1 0 0 1 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
        0)
    ],
    [
        E: (0 0 0 0 0 0 0 0 0 0 0 1 2 0 0 0 0 0 2 0 1 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0
        0)
    ]
]
```

110.3.2 Groups of Lie Type

`TrivialRepresentation(G)`

The one-dimensional trivial representation of the group of Lie type G over its base ring.

`StandardRepresentation(G)`

The standard (projective) representation of the semisimple group of Lie type G over an extension its base ring. In other words, the smallest dimension highest-weight representation. For the classical groups, this is the natural representation. If this is a projective representation rather than a linear representation, a warning is given. This is constructed from the corresponding Lie algebra representation, using the algorithm in [CMT04].

`AdjointRepresentation(G)`

The adjoint (projective) representation of the group of Lie type G over an extension of its base ring, ie. the representation given by the action of G on its Lie algebra. The Lie algebra itself is the second returned value. This is constructed from the corresponding Lie algebra representation, using the algorithm in [CMT04].

`LieAlgebra(G)`

The Lie algebra of the group of Lie type G, together with the adjoint representation. If this is a projective representation rather than a linear representation, a warning is given.

`HighestWeightRepresentation(G, v)`

The highest weight (projective) representation with highest weight v of the group of Lie type G over an extension of its base ring. If this is a projective representation rather than a linear representation, a warning is given. This is constructed from the corresponding Lie algebra representation, using the algorithm in [CMT04].

Example H110E5

```
> G := GroupOfLieType("A2", Rationals() : Isogeny := "SC");
> rho := StandardRepresentation(G);
> rho(elt< G | 1 >);
[ 0 -1  0]
[ 1  0  0]
[ 0  0  1]
> rho(elt<G | <2,1/2> >);
[  1   0   0]
[  0   1   0]
[  0 1/2   1]
> rho(elt< G | VectorSpace(Rationals(),2)![3,5] >);
[  3   0   0]
[  0 5/3   0]
[  0   0 1/5]
>
> G := GroupOfLieType("A2", Rationals());
> Invariants(CoisogenyGroup(G));
[ 3 ]
> rho := StandardRepresentation(G);
Warning: Projective representation
> BaseRing(Codomain(rho));
Algebraically closed field with no variables
> rho(elt< G | VectorSpace(Rationals(),2)![3,1] >);
[r1  0  0]
[ 0 r2  0]
[ 0  0 r2]
> rho(elt< G | VectorSpace(Rationals(),2)![3,1] >)^3;
```

```
[  9   0   0]
[  0 1/3   0]
[  0   0 1/3]
```

110.4 Operations on Weight Multisets

110.4.1 Basic Operations

In this section, basic access and arithmetic operations for weight multisets are described. Addition generally corresponds to direct sum of representations. The other arithmetic operations do not necessarily correspond to meaningful operations on the corresponding representation.

`RootDatum(D)`

The Root datum over which the weight multiset D is defined.

`Weights(D)`

`WeightsAndMultiplicities(D)`

The sequences of weights and multiplicities in the weight multiset D.

`Multiset(D)`

The weights and multiplicities of the weight multiset D as a normal multiset consisting of vectors.

`Multiplicity(D, v)`

The multiplicity of the weight v in the weight multiset D.

`D eq E`

Returns `true` if, and only if, the weight multisets D and E are identical, i.e. they are defined over identical root data, with equal weights and multiplicities.

`D + E`

The sum (union) of weight multisets D and E, i.e. this corresponds to the direct sum of the two decomposition multisets. The underlying root data must be the same.
Add V_v to D. The length of v must be equal to $\dim(R_D)$.

`D +:= E`

Add the weight multiset E to D. R_D must be equal to R_E.

`AddRepresentation(~D, E, c)`

`AddRepresentation(~D, E)`

Add c times the weight multiset E to D. The integer c may be omitted, in which case it is assumed to be equal to 1. The root data of D and E must be identical.

`D + v`

Add the weight v to the multiset D. The weight v must be a sequence of length d or an element of $\mathbf{Z}^d$, where d is the dimension of the root datum R.

`AddRepresentation(~D, v, c)`

`AddRepresentation(~D, v)`

Add c times the weight v to the multiset D. The integer c may be omitted, in which case it is assumed to be equal to 1. The length of v must be equal to the dimension of the root datum of D.

`D +:= v`

Add the weight v to the multiset D. The length of v must be equal to the dimension of the root datum of D.

`D * c`

The multiset whose weights are equal to those of D, and whose multiplicities are c times the multiplicities of D.

`D / c`

The multiset whose weights are equal to those of D, and whose multiplicities are the multiplicities of D divided by c. An error is flagged if any of the multiplicities of D is not divisible by c.

`D *:= c`

Multiply all multiplicities of the weight multiset D by c.

`D /:= c`

Divide all multiplicities of the weight multiset D by c. An error is flagged if a multiplicity of D is not divisible by c.

`D * E`

`ProductRepresentation(D, E)`

The product of the two weight multisets D and E, viewed as polynomials as in the LiE package [vLCL92]. The root datum of the resulting decomposition is the direct sum of the root data of D and E. Note that this is does not correspond to the direct sum or tensor product of representations.

`ProductRepresentation(D, E, R)`

The product of the two weight multisets D and E, viewed as polynomials as in the LiE package [vLCL92]. The product is interpreted as a weight multiset over the root datum R. An error is flagged if the dimension of R is not the sum of the dimensions of the root data of D and E.

`SubWeights(D, Q, S)`

Let k be the length of the sequence Q. The resulting decomposition E has Root datum S, and to each highest weight of D corresponds a highest weight w' of E, with $w'_i = w_{Q[i]}$, where $i = 1, \ldots, k$. The multiplicities of E are equal to the multiplicities of D, but one should note that E might in fact have fewer unique highest weights than D, especially if $k < \dim(R_D)$. The dimension of the root datum S must be equal to k.

`PermuteWeights(D, pi, S)`

Permute the components of the weights in the multiset D by the permutation π and interpret the result as a weight multiset over the root datum S. If the underlying root datum of D has dimension d, then S must also have dimension d and π must be an element of $\mathrm{Sym}(d)$.

Example H110E6

Arithmetic with decompositions:

```
> R := RootDatum("A2" : Isogeny := "SC");
> D := LieRepresentationDecomposition(R, [[2,3],[4,3]], [1,3]);
> D:Maximal;
Highest weight decomposition of representation of:
     R: Simply connected root datum of dimension 2 of type A2
     Dimension of weight space:2
     Weights:
          [
               (2 3),
               (4 3)
          ]
     Multiplicities:
          [ 1, 3 ]
> E := D + [5,2];
> E:Maximal;
Highest weight decomposition of representation of:
     R: Simply connected root datum of dimension 2 of type A2
     Dimension of weight space:2
     Weights:
          [
               (2 3),
               (4 3),
               (5 2)
```

```
          ]
     Multiplicities:
          [ 1, 3, 1 ]
> PermuteWeights(E, Sym(2)!(1,2), R):Maximal;
Highest weight decomposition of representation of:
     R: Simply connected root datum of dimension 2 of type A2
     Dimension of weight space:2
     Weights:
          [
               (3 2),
               (3 4),
               (2 5)
          ]
     Multiplicities:
          [ 1, 3, 1 ]
> S := RootDatum("A1" : Isogeny := "SC");
> SubWeights(E, [2], S):Maximal;
Highest weight decomposition of representation of:
     S: Simply connected root datum of dimension 1 of type A1
     Dimension of weight space:1
     Weights:
          [
               (3),
               (2)
          ]
     Multiplicities:
          [ 4, 1 ]
```

110.4.2 Conversion Functions

Functions for converting between different kinds of weight multiset (decomposition, character, and dominant character multisets). Note that it is the users responsibility to keep track of what kind of multiset they are using. If a function that expects one kind of set receives another, the output is likely to be meaningless.

`VirtualDecomposition(C)`

`VirtualDecomposition(R, v)`

The virtual decomposition multiset of the virtual module with dominant character multiset C. The second version is provided for convenience, and equivalent to `VirtualDecomposition(LieRepresentationDecomposition(R,v))`.

`DecomposeCharacter(C)`

The decomposition multiset of the module with dominant character multiset C. An error is flagged if D is virtual, i.e. if dominant weights occur with negative multiplicities.

`DominantCharacter(D)`

Returns the dominant character multiset with decomposition D.

110.4.3 Calculating with Representations

As described earlier, many operations on representations carry over naturally to operations on their decompositions. This section describes the various functions for this purpose that were ported from LiE.

Note that many functions in this sections have two variants: one that takes decompositions as an argument and one that takes a root datum and a highest weight.

`RepresentationDimension(D)`

The dimension of the module with decomposition polynomial D. The algorithm used is described in [vLCL92].

`RepresentationDimension(R, v)`

The dimension of the module with highest weight v over the root datum R. The algorithm used is described in [vLCL92].

`CasimirValue(R, w)`

The value of the quadratic Casimir on representation with highest weight w, normalised to take the value 2 on the highest weight of the adjoint representation. This function is due to Dr. Bruce Westbury, University of Warwick.

`QuantumDimension(R, w)`

Two Multisets of positive integers, Num and Den, which should be read as follows. Take the product of the integers in Num and divide by the product of the integers in Den to get the ordinary dimension. Replacing each integer by the quantum integer will give the quantum dimension. This function is due to Dr. Bruce Westbury, University of Warwick.

Example H110E7

Dimensions:

```
> R := RootDatum("D4" : Isogeny := "SC");
> D := AdjointRepresentationDecomposition(R);
> RepresentationDimension(D);
28
> wts, mps := WeightsAndMultiplicities(D); wts,mps;
[
    (0 1 0 0)
]
[ 1 ]
> num,den := QuantumDimension(R, wts[1]); num,den;
{* 4^^2, 7 *}
{* 1, 2^^2 *}
```

```
> &*num/&*den;
28
```

`Branch(FromGrp, ToGrp, v, M)`

`Virtual` BoolElt *Default :* `false`

The decomposition polynomial of the restriction to ToGrp of the irreducible module V_v with respect to the restriction matrix M. The matrix M must have dim(FromGrp) rows and Dim(ToGrp) columns.

The matrix M is used in such a way that any weight v' (expressed on the basis of fundamental weights for g), when restricted to a torus of ToGrp, becomes the weight $v'M$ (expressed on the basis of fundamental weights for ToGrp). A suitable restriction matrix can often be obtained by use of `RestrictionMatrix`. The algorithm used is described in [vLCL92].

The optional argument `Virtual` may be set to `true` to allow occurrence of virtual weights.

`Branch(ToGrp, D, M)`

`Virtual` BoolElt *Default :* `false`

As `Branch(FromGrp, ToGrp, v, M)` but with the irreducible module v replaced by the module with decomposition D.

`Collect(R, D, M)`

This function attempts to perform the inverse operation of `Branch`, namely to reconstruct an R-module from its restriction to R_D.

Please note that in LiE one must supply the inverse of the matrix used in Branch. Magma, however, is able to compute inverses itself, so one needs to provide the matrix used in Branch, and not its inverse.

M must be a square matrix whose dimension is equal to the dimension of R_D. The dimension of R must be equal to the dimension of R_D as well. The algorithm used is described in [vLCL92].

Example H110E8

Branch and Collect:

```
> R := RootDatum("D4" : Isogeny := "SC");
> S := RootDatum("A3T1" : Isogeny := "SC");
> M := RestrictionMatrix(R, S);
> br := Branch(R, S, [1,0,0,0], M);
> br;
Highest weight decomposition of representation of:
     S: Simply connected root datum of dimension 4 of type A3
     Number of terms: 2
> cl := Collect(R, br, M);
```

```
> cl:Maximal;
Highest weight decomposition of representation of:
     R: Simply connected root datum of dimension 4 of type D4
     Dimension of weight space:4
     Weights:
          [
               (1 0 0 0)
          ]
     Multiplicities:
          [ 1 ]
```

`TensorProduct(R, v, w)`

`Goal` ANY *Default :*

The decomposition multiset of the tensor product of the representations with highest weights v and w over the root datum R.

If the optional parameter `Goal` is set, only the multiplicity of the irreducible module with highest weight `Goal` is returned. This does not greatly speed up the process, as the same computational steps need to be made, but it will significantly reduce memory consumption. The algorithm used is described in [vLCL92].

`TensorProduct(D, E)`

`Goal` ANY *Default :*

The decomposition multiset of the tensor product of the representations with decomposition multisets D and E.

If the optional parameter `Goal` is set, only the multiplicity of the irreducible module with highest weight `Goal` is returned. This does not greatly speed up the process, as the same computational steps need to be made, but it will significantly reduce memory consumption. The algorithm used is described in [vLCL92].

`TensorProduct(Q)`

`Goal` ANY *Default :*

The decomposition multiset of the tensor product of the representations with decomposition multisets in the sequence Q.

If the optional parameter `Goal` is set, only the multiplicity of the irreducible module with highest weight `Goal` is returned. This does not greatly speed up the process, as the same computational steps need to be made, but it will significantly reduce memory consumption. The algorithm used is described in [vLCL92].

`TensorPower(R, n, v)`

`TensorPower(D, n)`

The decomposition of the n-th tensor power of V_v^R or D.

Example H110E9

Taking tensor powers nicely shows how rapidly the complexity of representations increases, especially if we have a reasonably high weight as highest weight:

```
> R := RootDatum("D4" : Isogeny := "SC");
> DAd := AdjointRepresentationDecomposition(R);
> pwrs := function(D, n)
>   Q := [D];
>   for i in [2..n] do
>     Q[i] := Tensor(Q[1], Q[i-1]);
>   end for;
>   return Q;
> end function;
> time Q := pwrs(DAd, 7);
Time: 4.900
> [ #q : q in Q ];
[ 1, 7, 15, 30, 54, 91, 143 ]
> DH := LieRepresentationDecomposition(R, [2,2,0,0]);
> time Q := pwrs(DH, 4); [ #q : q in Q ];
Time: 99.070
[ 1, 105, 390, 1017 ]
```

`AdamsOperator(R, n, v)`

`AdamsOperator(D, n)`

The decomposition polynomial of the virtual module obtained by applying the n-th Adams operator to V_v^R or D. The algorithm used is described in [vLCL92].

`SymmetricPower(R, n, v)`

`SymmetricPower(D, n)`

The decomposition polynomial of $S^n(V_v^R)$, the n-th symmetric tensor power of V_v^R.

In the second form the irreducible module V_v^R is replaced by the module with decomposition D. The algorithm used is described in [vLCL92].

`AlternatingPower(R, n, v)`

`AlternatingPower(D, n)`

The decomposition polynomial of $\mathrm{Alt}^n(V_v^R)$, the n-th alternating tensor power of V_v^R.

In the second form the irreducible module V_v^R is replaced by the module with decomposition D. The algorithm used is described in [vLCL92].

`Plethysm(R, lambda, v)`

`Plethysm(D, lambda)`

The decomposition multiset of the R_D-module of the plethysm of V_v^R corresponding to the partition λ. Here λ should be a partition of $d = \dim V_v^R$, i.e., a non-increasing sequence consisting of positive integers with sum d. The value returned is the decomposition multiset of the representation of R_D that is obtained by composing the representation of R_D afforded by V_v^R, with the representation of $\mathrm{GL}(V_v^R)$ corresponding to the partition λ. The classical Frobenius formula is used (see [And77] and [JK81]).

In the second form the irreducible module V_v^R is replaced by the module with decomposition D.

`Spectrum(R, v, t)`

`Spectrum(D, t)`

Let n be the last entry of the sequence t; the toral element $t \in T$ will act in any representation of R as a diagonalisable transformation, all of whose eigenvalues are n-th roots of unity. This function returns a sequence in which the i-th entry is the multiplicity of the eigenvalue ζ^i in the action of the toral element t on the irreducible module V_v^R (or the module with decomposition D, in the second case). Here ζ is the complex number $e^{2\pi i/n}$.

See Section 110.1.2 for a description of the format of t.

Example H110E10

Spectrum provides a means to recognise toral elements in a more natural form. [vLCL92, Section 5.7.3].

```
> R := RootDatum("A4" : Isogeny := "SC");
> stdrep := [1,0,0,0];
> t := [1,0,0,0,2];
> stdrep := [1,0,0,0];
> Spectrum(R, stdrep, t);
[ 3, 2 ]
/* Showing that t has 3 eigenvalues 1 (1st root of unity),
   and 2 eigenvalues -1 (2nd root of unity) */
/* We may use the following function for constructing
   toral elements of A_n in the LiE format: */
> mktoral := function(b, d)
>     r := [ (i eq 1)
>                 select b[i]
>                 else b[i-1]+b[i] mod d
>               : i in [1..(#b-1)]
>          ];
>     r[#b] := d;
>     return r;
```

```
> end function;
> t2 := mktoral([0,0,0,1,1], 2); t2;
[ 0, 0, 0, 1, 2 ]
/* We restrict to a one parameter subgroup */
> RM := Transpose(Matrix([[0,0,0,1]]));
> T1 := RootDatum("T1" : Isogeny := "SC");
> Branch(R, T1, stdrep, RM):Maximal;
Highest weight decomposition of representation of:
     T1: Toral root datum of dimension 1
     Dimension of weight space:1
     Weights:
          [
               (1),
               (0),
               (-1)
          ]
     Multiplicities:
          [ 1, 3, 1 ]
/* Indicating that the element of that one parameter
   subgroup parametrised by some complex number z has
   one eigenvalue z^-1, three eigenvalues 1, and one
   eigenvalue z in the standard representation. */
```

`Demazure(R, v, w)`

`Demazure(D, w)`

Starting with the highest weight v of R, or the decomposition D, repeatedly apply the Demazure operator M_{α_i}, taking for i the successive entries of the Weyl word w (viewed as product of simple reflections).

`Demazure(R, v)`

`Demazure(D)`

Equivalent to `Demazure(R, v, w)` or `Demazure(D, w)` where w is the longest word of the Coxeter group of R or R_D.

If D is a decomposition polynomial, then the result E is the character polynomial of this decomposition. This is not the most efficient way to compute characters, but it can be very useful in checking other algorithms, since only the most elementary manipulations are involved.

Example H110E11

The Demazure operator:

```
> R := RootDatum("D4" : Isogeny := "SC");
> DAd := AdjointRepresentationDecomposition(R);
> DAdCp := Demazure(DAd); DAdCp;
Highest weight decomposition of representation of:
     R: Simply connected root datum of dimension 4 of type D4
     Number of terms: 25
> DAd2 := AlternatingDominant(DAdCp); DAd2;
Highest weight decomposition of representation of:
     R: Simply connected root datum of dimension 4 of type D4
     Number of terms: 1
> DAd2 eq DAd;
true
```

`LittlewoodRichardsonTensor(p, q)`

`LittlewoodRichardsonTensor(P, M, Q, N)`

In the first form, p and q are interpreted as dominant weights for the group SL_n (of type A_{n-1}) expressed in partition coordinates. Here n is the number of elements of p (which must be equal to the number of elements of q).

The tensor product of the corresponding highest weight modules is computed using the Littlewood-Richardson rule, and the result is expressed again in partition coordinates. To be precise, two sequences P, M, are returned, meaning that the highest weight module with partition coordinates P[i] occurs in the tensor product with multiplicity M[i].

In the second form, instead of two irreducible modules, the tensor product of the module having partition coordinates P[i] with multiplicity M[i] and the module having partition coordinates Q[j] with multiplicity N[j] is computed.

`LittlewoodRichardsonTensor(R, v, w)`

`LittlewoodRichardsonTensor(D, E)`

In the first form, compute the tensor product of the irreducible A_n representations with highest weights v and w using the Littlewood-Richardson rule. In the second form, compute the tensor product of the representations with decompositions D and E.

This procedure converts the weights to partitions, computes the tensor product using the Littlewood-Richardson rule (as described above, see LittlewoodRichardsonTensor), and converts the result back to a weight multiset.

Example H110E12

We compare the Littlewood-Richardson tensor and the normal tensor:

```
> R := RootDatum("A2" : Isogeny := "SC");
> v := [1,2];
> w := [1,1];
> D1 := Tensor(R, v, w);
> D1;
Highest weight decomposition of representation of:
    R: Simply connected root datum of dimension 2 of type A2
    Weights:
        [
            (0 1),
            (2 0),
            (0 4),
            (3 1),
            (2 3),
            (1 2)
        ]
    Multiplicities:
        [ 1, 1, 1, 1, 1, 2 ]
> D2 := LittlewoodRichardsonTensor(R, v, w);
> D2;
Highest weight decomposition of representation of:
    R: Simply connected root datum of dimension 2 of type A2
    Weights:
        [
            (1 2),
            (2 3),
            (2 0),
            (0 4),
            (0 1),
            (3 1)
        ]
    Multiplicities:
        [ 2, 1, 1, 1, 1, 1 ]
> D1 eq D2;
true
```

So the results are identical, as they should be. We could also convert the weights to partitions by hand, directly compute the Littlewood- Richardson tensor, and compare that to the previous result:

```
> vp := WeightToPartition(v); wp := WeightToPartition(w);
> vp, wp;
[ 3, 2, 0 ]
[ 2, 1, 0 ]
> parts, mps := LittlewoodRichardsonTensor(vp, wp);
> parts, mps;
```

```
[
    (4 4 0),
    (4 3 1),
    (3 3 2),
    (5 3 0),
    (5 2 1),
    (4 2 2)
]
[ 1, 2, 1, 1, 1, 1 ]
> [ PartitionToWeight(p) : p in parts ];
[
    (0 4),
    (1 2),
    (0 1),
    (2 3),
    (3 1),
    (2 0)
]
```

So that again gives the same representation. Finally, note that in some cases computing tensor products using the Littlewood-Richardson rule may be faster than computing them in the normal way:

```
> R := RootDatum("A8" : Isogeny := "SC");
> v := [0,0,2,0,1,0,1,2];
> w := [0,2,1,2,0,0,1,0];
> time _ := Tensor(R, v, w);
Time: 2.630
> time _ := LittlewoodRichardsonTensor(R, v, w);
Time: 0.210
```

`AlternatingDominant(D, w)`

`AlternatingDominant(R, wt, w)`

Alternating Dominant of the representation with decomposition D or the irreducible representation V_{wt}, with respect to Weyl group element w. Starting with D, the following operation is repeatedly applied, taking for i the successive entries of w (viewed as reflection). For any (weight, multiplicity) pair (v, c) of D let $v_i = \langle v, \alpha_i \rangle$ be its coefficient of w_i; the term is

- unaltered if $v_i \geq 0$,
- removed if $v_i = -1$, and
- replaced by $((v + w_i)r_i - w_i, -c)$ if $v_i = -2$. As a result of the operation for i, the coefficient v_i is made non-negative without affecting the image $M_{\alpha_i}(D)$ under the Demazure operator, and hence also without changing the value of its alternating Weyl sum `AlternatingWeylSum`.

`AlternatingDominant(D)`

`AlternatingDominant(R, wt)`

Equivalent to (but somewhat faster than) the previous `AlternatingDominant`(D, w) and `AlternatingDominant`(R, wt, w), with w the longest element of the corresponding Weyl group. If D is interpreted as dominant weights with multiplicities, then the result E contains highest weights and multiplicities.

Example H110E13

Example of the alternating dominant:

```
> R := RootDatum("D4" : Isogeny := "SC");
> v := [1,5,2,1];
> Dec1 := LieRepresentationDecomposition(R, v);
> // First, we construct the character polynomial for the
> // module with highest weight lambda
> Dom := DominantCharacter(Dec1 : InBasis := "Weight"); Dom;
Highest weight decomposition of representation of:
     R: Simply connected root datum of dimension 4 of type D4
     Number of terms: 176
> W := CoxeterGroup(R); #W; act := RootAction(W);
192
> domwts, dommps := WeightsAndMultiplicities(Dom);
> CP := LieRepresentationDecomposition(R);
> for i in [1..#domwts] do
>   wt := domwts[i]; mp := dommps[i];
>   wtor := WeightOrbit(W, wt : Basis := "Weight");
>   for wti in wtor do
>     AddRepresentation(~CP, wti, mp);
>   end for;
> end for;
> CP;
Highest weight decomposition of representation of:
     R: Simply connected root datum of dimension 4 of type D4
     Number of terms: 17712
> time ad := AlternatingDominant(CP); ad:Maximal;
Time: 54.200
Highest weight decomposition of representation of:
     R: Simply connected root datum of dimension 4 of type D4
     Dimension of weight space:4
     Weights:
          [
               (1 5 2 1)
          ]
     Multiplicities:
          [ 1 ]
> time adalt := AlternatingDominant(CP, LongestElement(W));
```

```
Time: 8.330
> ad eq adalt;
true
```

`AlternatingWeylSum(R, v)`

`AlternatingWeylSum(D)`

The alternating Weyl sum of V_v^R or D. Useful for demonstration purposes, but the fact that the number of terms in the result is a multiple of the order of the CoxeterGroup of R makes it impractical for most groups.

110.5 Operations on Representations

110.5.1 Lie Algebras

The functions described in this section are applicable only to modules of almost reductive structure constant Lie algebras.

`CharacterMultiset(V)`

`CharacterMultiset(ρ)`

The character multiset of the Lie-algebra module V or representation ρ.

`Weights(V)`

`WeightsAndVectors(V)`

For a module V over a semisimple Lie algebra this returns two sequences. The first sequence consists of the weights that occur in V. The second sequence is a sequence of sequences of elements of V, in bijection with the first sequence. The i-th element of the second sequence consists of a basis of the weight space of weight equal to the i-th weight of the first sequence.

`Weights(ρ)`

`WeightsAndVectors(ρ)`

For a representation ρ of a semisimple Lie algebra this returns two sequences. The first sequence consists of the weights of ρ. The second sequence is a sequence of sequences of elements of the underlying vector space, in bijection with the first sequence. The i-th element of the second sequence consists of a basis of the weight space of weight equal to the i-th weight of the first sequence.

`DecompositionMultiset(V)`

`DecompositionMultiset(ρ)`

The decomposition multiset of the Lie-algebra module V or representation ρ.

`HighestWeightsAndVectors(V)`

This function is analogous to the previous one. Except in this case the first sequence consists of highest weights, i.e., those weights which occur as highest weights of an irreducible constituent of V. The second sequence consists of sequences that contain the corresponding highest weight vectors. So the submodules generated by the vectors in the second sequence form a direct sum decomposition of V.

`DirectSum(U, V)`

The direct sum of the Lie algebra modules U and V.

`DirectSumDecomposition(V)`

`IndecomposableSummands(V)`

Given a Lie algebra module V, return the direct sum decomposition of V as a sequence of submodules whose sum is V and each of which cannot be further decomposed into a direct sum. If the Lie algebra is semisimple over a field of characteristic zero, the summands are known to be irreducible highest weight modules.

`DirectSum(`ρ`, `τ`)`

The direct sum of the Lie algebra representations ρ and τ.

`DirectSumDecomposition(`ρ`)`

`IndecomposableSummands(`ρ`)`

Given a Lie algebra representation ρ, return the direct sum decomposition of ρ as a sequence of indecomposable subrepresentation. If the Lie algebra is semisimple over a field of characteristic zero, the summands are known to be irreducible highest weight representations.

`TensorProduct(Q)`

Given a sequence Q of left-modules over a Lie algebra, this function returns the module M that is the tensor product of the elements of Q. Secondly it returns a map from the Cartesian product of the elements of Q to M. This maps a tuple t to the element of M that is formed by tensoring the elements of t.

`SymmetricPower(V, n)`

Given a left-module V over a Lie algebra, and an integer $n \geq 2$, this function returns the module M that is the n-th symmetric power of V. It also returns a map f from the n-fold Cartesian product of V to M. This map is multilinear and symmetric, i.e., if two of its arguments are interchanged then the image remains the same. Furthermore, f has the universal property, i.e., any multilinear symmetric map from the n-fold Cartesian product into a vector space W can be written as the composition of f with a map from M into W.

`ExteriorPower(V, n)`

Given a left-module V over a Lie algebra, and an integer $2 \leq n \leq \dim(V)$, this function returns the module M that is the n-th exterior power of V. It also returns a map f from the n-fold Cartesian product of V to M. This map is multilinear and antisymmetric, i.e., if two of its arguments are interchanged then the image is multiplied by -1. Furthermore, f has the universal property, i.e., any multilinear antisymmetric map from the n-fold Cartesian product into a vector space W can be written as the composition of f with a map from M into W.

Example H110E14

```
> L:= LieAlgebra("G2", Rationals());
> V1:= HighestWeightModule(L, [1,0]);
> V2:= HighestWeightModule(L, [0,1]);
> T,f:= TensorProduct([V1,V2]);
> HighestWeightsAndVectors(T);
[
    (1 1),
    (2 0),
    (1 0)
]
[
    [
        T: (1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
            0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
            0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0)
    ],
    [
        T: (0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 3 0 0 0 0 0 0 0 0 0 0 0 0 -3 0 0 0 0 0
            0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
            0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0)
    ],
    [
        T: (0 0 0 0 0 0 1 2 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 -1 0 0
            0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 -3 0 0 0 0 0 0 0 0 0 0
            0 0 -3 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0)
    ]
]
> DecomposeTensorProduct(RootDatum(L), [1,0], [0,1]);
[
    (1 1),
    (2 0),
    (1 0)
]
[ 1, 1, 1 ]
> f(<V1.2,V2.3>);
T: (0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
```

```
    0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
    0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0)
```

So we see that the tensor product T decomposes as a direct sum of three submodules. This information can also be computed by using `DecomposeTensorProduct`. However, in the former case, the corresponding highest-weight vectors are also given.

```
> E,h:= ExteriorPower(V1, 3);
> h(<V1.1,V1.3,V1.4>);
E: (0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0)
> h(<V1.1,V1.4,V1.3>);
E: (0 0 0 0 0 -1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0)
> DecomposeExteriorPower( RootDatum(L), 3, [1,0] );
[
    (2 0),
    (1 0),
    (0 0)
]
[ 1, 1, 1 ]
> HighestWeightsAndVectors(E);
[
    (2 0),
    (1 0),
    (0 0)
]
[
    [
        E: (1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
        0)
    ],
    [
        E: (0 0 0 1 0 0 1 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
        0)
    ],
    [
        E: (0 0 0 0 0 0 0 0 0 0 0 1 2 0 0 0 0 0 2 0 1 0 0 0 0 1 0 0 0 0 0 0 0 0
        0)
    ]
]
```

110.5.2 Groups of Lie Type

These functions apply to projective representations of groups of Lie type. Note that modules have not yet been implemented for these groups.

`DirectSum(ρ, τ)`

The direct sum of the group of Lie type representations ρ and τ.

`DirectSumDecomposition(ρ)`
`IndecomposableSummands(ρ)`

Given a group of Lie type representation ρ, return the direct sum decomposition of ρ as a sequence of indecomposable subrepresentation. If the base field has characteristic zero, the summands are known to be irreducible highest weight representations.

`CharacterMultiset(V)`
`CharacterMultiset(ρ)`

The character weight multiset of the group of Lie type representation ρ.

`Weights(ρ)`
`WeightsAndVectors(ρ)`

The weights of the representation ρ, together with the corresponding weight vectors.

`WeightVectors(ρ)`

A basis of weight vectors of the representation ρ.

`Weight(ρ, v)`

The weight corresponding to the weight vector v of the representation ρ.

`DecompositionMultiset(V)`
`DecompositionMultiset(ρ)`

The decomposition multiset of the group of Lie type representation ρ.

`HighestWeights(ρ)`

The highest weights of the representation ρ, together with the corresponding highest weight vectors. This function may fail for small finite fields.

`HighestWeightVectors(ρ)`

The highest weight vectors of the representation ρ.

`GeneralisedRowReduction(ρ)`

Given a projective matrix representation $\rho : G \to \mathrm{GL}_m(k)$, return its inverse. This algorithm is based on [CMT04].

110.6 Other Functions for Representation Decompositions

In this section, we describe more complicated functions for dealing with representation decompositions.

`FundamentalClosure(R, S)`

A set of fundamental roots in the minimal subsystem (not necessarily closed!) of the root system R that contains all the roots in S. This function is equivalent to the `fundam` function in LiE.

The set S should contain either roots or root indices; the returned set will then contain objects of the same type.

`Closure(R, S)`

A set of fundamental roots in the minimal subsystem (not necessarily closed!) of the root system R that contains all the roots in S. This function is equivalent to the `closure` function in LiE.

The set S should contain either roots or root indices; the returned set will then contain objects of the same type.

`RestrictionMatrix(R, Q)`

For a simply connected root datum R and a sequence of roots Q forming a fundamental basis for a closed subdatum S of R, this function computes a restriction matrix for the fundamental Lie subgroup of type S of the Lie group corresponding to R.

The sequence Q may contain either integers (where i corresponds to the i-th root of R) or vectors (interpreted as root vectors written in the root basis of R).

Note that the result is not unique. Moreover, if the result is to be used by Branch or Collect the roots in Q must be positive roots, and their mutual inner products must be non-positive.

`RestrictionMatrix(R, S)`

Let S be a sub root datum of R, constructed for example (but not necessarily) using a call to `sub<...>`. Then the matrix M returned by this function maps the fundamental weights of R to those of S. Note that, if the rank of S is smaller than the rank of R, there will be more than one such matrix.

Example H110E15

Constructing a restriction matrix

```
> R := RootDatum("D4":  Isogeny := "SC");
> sub<R | [1,3,4]>;
Root datum of dimension 4 of type A1 A1 A1
[ 1, 3, 4, 13, 15, 16 ]
> S := RootDatum("A1A1A1T1" : Isogeny := "SC");
> M := RestrictionMatrix(R, S); M;
[ 1  0  0 -1]
```

```
[ 0  1  0 -2]
[ 0  0  1 -1]
[ 0  0  0  4]
> imgR := FundamentalWeights(R)*M; imgR;
[ 1  0  0 -1]
[ 0  1  0 -2]
[ 0  0  1 -1]
[ 0  0  0  4]
> FundamentalWeights(S);
[1 0 0 0]
[0 1 0 0]
[0 0 1 0]
/* M is of the required form, since: */
> [ BasisChange(S, BasisChange(S, imgR[i]
>     : InBasis := "Standard", OutBasis := "Weight")
>     : InBasis := "Weight", OutBasis := "Standard")
>   : i in [1..4]
> ];
[
    (1 0 0 0),
    (0 1 0 0),
    (0 0 1 0),
    (0 0 0 0)
]
```

`KLPolynomial(x, y)`

`Ring` RngUPol *Default :* $\mathbf{Z}$[X]

The Kazhdan-Lusztig polynomial $P_{x,y}$. We use the recursion given originally by Kazhdan and Lusztig [KL79].

`RPolynomial(x, y)`

`Ring` RngUPol *Default :* $\mathbf{Z}$[X]

The R-polynomial $R_{x,y}$.

Example H110E16

There is a relation between Kazhdan-Lusztig polynomials and R-polynomials. We should have, for any $x, w \in W$:

$$X^{l(w)-l(x)}\overline{P_{x,w}} - P_{x,w} = \sum_{x<y\leq w} R_{x,w}P_{y,w},$$

where the bar indicates a sign change of all the exponents. We need some fiddling around in order to implement this sign change, since Magma doesn't support negative exponents at the moment, but we can make it work:

```
> signchange := function(pol, pwr)
```

```
>     //returns X^pwr * bar(pol)
>     deg := Degree(pol);
>     P := Parent(pol);
>     if (deg gt pwr) then return "Failed: Can't do sign change"; end if;
>     return (P.1)^(pwr-deg)*P!Reverse(Eltseq(pol));
> end function;
> testKL := function(x, w)
>     W := Parent(x);
>     rng<X> := PolynomialRing(Integers());
>     lenw := CoxeterLength(W, w);
>     lenx := CoxeterLength(W, x);
>
>     if (lenx gt lenw) then
>         return "Failed: l(x) > l(w) gives zero R and KL polynomials.";
>     end if;
>
>     /* Left hand side */
>     Pxw := KLPolynomial(x, w : Ring := rng);
>     lhs := signchange(Pxw, lenw - lenx);
>     if (Type(lhs) eq MonStgElt) then return lhs; end if;
>     lhs -:= Pxw;
>
>     /* Right hand side */
>     rhs := rng!0;
>     lvl := {w};
>     lvllen := lenw;
>     while (lvllen gt lenx and #lvl gt 0) do
>         for y in lvl do
>             rhs +:= RPolynomial(x,y : Ring := rng)*
>                         KLPolynomial(y, w : Ring := rng);
>         end for;
>         lvl := BruhatDescendants(lvl : z := x);
>         lvllen -:= 1;
>     end while;
>
>     /* Done */
>     printf "LHS: %o\n", lhs;
>     printf "RHS: %o\n", rhs;
>     return lhs eq rhs;
> end function;
> W := CoxeterGroup("D4");
> x := W.1*W.2*W.1;
> w := W.1*W.2*W.3*W.4*W.1*W.2;
> testKL(x,w);
LHS: X^3 + X^2 - X - 1
RHS: X^3 + X^2 - X - 1
```

```
true
```

`Exponents(R)`

The exponents of a Root datum R form a sequence of numbers $e_1, \ldots, e_r$, where r is the rank of R, such that the polynomial $\sum_{w \in W} X^{l(w)}$ decomposes as a product $\prod_{i=1}^{r} \sum_{j=0}^{e_i} X^j$. They are given in weakly increasing order.

Example H110E17

Exponents of A_3:

```
> W := CoxeterGroup("A3"); #W;
24
> P<X> := PolynomialRing(Integers());
> f := &+[ X^(CoxeterLength(W, w)) : w in W ]; f;
X^6 + 3*X^5 + 5*X^4 + 6*X^3 + 5*X^2 + 3*X + 1
> R := RootDatum("A3" : Isogeny := "SC");
> exp := Exponents(R); exp;
[ 1, 2, 3 ]
> g := [ &+[ X^j : j in [0..e] ] : e in exp ]; g;
[
    X + 1,
    X^2 + X + 1,
    X^3 + X^2 + X + 1
]
> f eq &*g;
true
```

`ToLiE(D)`

The LiE equivalent of the decomposition D.

`FromLiE(R, p)`

The decomposition of the representation over R that is equivalent to p, where p is a polynomial in LiE-syntax.

Example H110E18

Conversion to and from LiE-syntax

```
> R := RootDatum("B3" : Isogeny := "SC");
> D := LieRepresentationDecomposition(R, [1,2,3]);
> s := ToLiE(D); s;
1X[1,2,3]
> FromLiE(R, s):Maximal;
Highest weight decomposition of representation of:
     R: Simply connected root datum of dimension 3 of type B3
     Dimension of weight space:3
     Weights:
          [
               (1 2 3)
          ]
     Multiplicities:
          [ 1 ]
```

110.6.1 Operations Related to the Symmetric Group

In this section, we describe some functions taken from LiE for dealing with the Symmetric group.

`ConjugationClassLength(l)`

The order of the conjugation class of S_n of permutations of cycle type l (for n the sum of the elements of l).

`PartitionToWeight(l)`

Let n be the number of parts of l, then the function returns the weight for a group of type A_{n-1} corresponding to λ, expressed on the basis of fundamental weights.

`WeightToPartition(v)`

Let n be the length of v, then v is interpreted as a weight for a group of type A_n, and the expression of that weight in $n+1$ partition coordinates is returned. When v is dominant, this is a partition with $n+1$ parts.

`TransposePartition(l)`

The transpose partition of l.

110.6.2 FusionRules

In this section, we describe a function for computing fusion rules using the Kac-Walton formular, as described in Section 16.2 of [FMS97].

`WZWFusion(R, v, w, k)`

`ReturnForm` MonStgElt *Default :* "*Auto*"

Compute the fusion rules for weights $v \times w$ of R at level k using the Kac-Walton formula. The weights v and w may be given either as finite weights (i.e. vectors or sequences with rank(R) entries) or as affine weights (i.e. vectors or sequences with rank(R)+1 entries).

The optional argument `ReturnForm` should be "Auto" (in which case the weights returned will be finite weights or affine weights depending on what v and w are; the default), "Finite" (in which case the weights returned are finite weights), or "Affine" (in which case the weights returned are affine weights).

Note that R should be a weakly simply connected root datum.

`WZWFusion(D, E, k)`

Compute the fusion rules for representations D and E at level k.

Example H110E19

Fusion rules at level 3 for B_3 (using finite weights first, affine weights second)

```
> R := RootDatum("B3" : Isogeny := "SC");
> WZWFusion(R, [0,0,1],[1,0,1], 3);
{*
    (2 0 0),
    (1 1 0),
    (0 0 2),
    (1 0 2),
    (1 0 0),
    (0 1 0)
*}
> WZWFusion(R, [0,0,1],[1,0,1], 3 : ReturnForm := "Affine");
{*
    (2 0 0 1),
    (1 0 2 0),
    (0 1 0 1),
    (1 0 0 2),
    (0 0 2 1),
    (1 1 0 0)
*}
```

110.7 Subgroups of Small Rank

LiE contains a small database with the types of the maximal proper subgroups of complex reductive simply connected Lie groups g, where g is simple and of rank at most 8. We copied this list into MAGMA, and it can be accessed using the following functions.

`LiEMaximalSubgroups()`

All maximal subgroups as described above, as a sequence of pairs. Each pair consists of a string denoting the simple group at hand, and a sequence of strings denoting its maximal subgroups.

`MaximalSubgroups(G)`

The maximal subgroups of the complex reductive simply connected simple Lie group whose Cartan type is the string G, represented as a sequence of strings.

`RestrictionMatrix(G, H)`

`Index` RNGINTELT *Default :*

The restriction matrix for the maximal proper subgroup of type H of G. If more than one maximal subgroup of G is of type H, the parameter `Index` must be set to indicate which one is required.

Example H110E20

Using the subgroup database:

```
> MaximalSubgroups("E7");
[ A2, A1, A1, A1F4, G2C3, A1G2, A1A1, D6A1, A7, A5A2 ]
> M := RestrictionMatrix("E7", "A1" : Index := 2); M;
[26]
[37]
[50]
[72]
[57]
[40]
[21]
> R := RootDatum("E7" : Isogeny := "SC");
> S := RootDatum("A1" : Isogeny := "SC");
> D := AdjointRepresentationDecomposition(R);
> RepresentationDimension(D);
133
> E := Branch(S, D, M); #E;
8
> RepresentationDimension(E);
133
```

110.8 Subalgebras of su(d)

This section describes functions for studying irreducible simple subalgebras of the Lie algebra $\mathfrak{su}(d)$ (cf. [Dyn57]). The verbose flag `"SubSU"` may be set to show details and progress of the various computations.

The algorithms and the implementation in this package are due to Robert Zeier. For more information about some of the algorithms used and the results obtained using this package we refer to [ZSH11].

`IrreducibleSimpleSubalgebrasOfSU(N)`

A list of all irreducible simple subalgebras occurring in the Lie algebra $\mathfrak{su}(d)$, for $2 \leq d \leq N$.

`IrreducibleSimpleSubalgebraTreeSU(Q, d)`

The subalgebra tree for degree d as a directed graph whose vertex labels describe subalgebras, derived from the list Q of irreducible subalgebras. The vertex labels are records with three fields: `algebra`, a string containing the Cartan type of this subalgebra; `weights`, a sequence of highest weights (as sparse vectors) corresponding to irreducible representations (they are related by an outer automorphism if there is more than one highest weight); and `type`, an integer with values -1, 1, or 0 corresponding to irreducible representations of quaternionic, real, or complex type, respectively (the Frobenius-Schur indicator).

`PrintTreesSU(Q, F)`

`FromDegree`	RngIntElt	*Default :* 2
`ToDegree`	RngIntElt	*Default :* $\lvert Q \rvert$
`IncludeTrivial`	BoolElt	*Default :* `true`

Print the tree of subalgebras in the sequence Q (as obtained by a call to `IrreducibleSimpleSubalgebrasOfSU`) to the file with filename F. The file F will be overwritten.

The resulting file will be a LaTeX document that may be typeset using `latex` followed by `dvipdf`, for instance. If the resulting file is large, the main memory allocated to TEX may have to be increased (the `main_memory` directive in `texmf.cnf`). Contact your system administrator in case of difficulty.

The optional arguments `FromDegree` and `ToDegree` limit which degrees are output; `IncludeTrivial` may be set to `false` to remove "trivial" cases (i.e. trivial trees) from the output. For $d \geq 5$ and d even, $\mathfrak{su}(d)$ is considered trivial if it contains only the (proper) irreducible simple subalgebras $C_{d/2}$ (i.e. $\mathfrak{sp}(d/2)$), $D_{d/2}$ (i.e. $\mathfrak{so}(d)$), and A_1 (i.e. $\mathfrak{su}(2)$); for $d \geq 5$ and d is odd, $\mathfrak{su}(d)$ is considered trivial if it contains only $B_{(d-1)/2}$ (i.e. $\mathfrak{so}(d)$) and A_1.

The Lie algebras in the output are coloured according to type: red for -1, blue for 1, and black for 0 (see `IrreducibleSimpleSubalgebraTreeSU`).

Example H110E21

We investigate subalgebras of $\mathfrak{su}(d)$ for d up to 2^{10}.

```
> Q := IrreducibleSimpleSubalgebrasOfSU(2^10);
> t := IrreducibleSimpleSubalgebraTreeSU(Q, 12);
> t;
Digraph
Vertex  Neighbours
1       2 4 ;
2       3 ;
3       ;
4       ;
> r := VertexLabel(t, 1); r`algebra;
rec<recformat<algebra: MonStgElt, weights, type: IntegerRing()> |
    algebra := A11,
    weights := [
        Sparse matrix with 1 row and 11 columns over Integer Ring,
        Sparse matrix with 1 row and 11 columns over Integer Ring
    ],
    type := 0>
> r := VertexLabel(t, 2); r;
rec<recformat<algebra: MonStgElt, weights, type: IntegerRing()> |
    algebra := C6,
    weights := [
        Sparse matrix with 1 row and 6 columns over Integer Ring
    ],
    type := -1>
> [ Matrix(w) : w in r`weights ];
[
    [1 0 0 0 0 0]
]
> RepresentationDimension(RootDatum("C6"),[1,0,0,0,0,0]);
12
> r := VertexLabel(t, 3); r`algebra;
A1
> [ Matrix(w) : w in r`weights ];
[
    [11]
]
> RepresentationDimension(RootDatum("A1"),[11]);
12
> r := VertexLabel(t, 4); r`algebra;
D6
```

In this manner we have used `IrreducibleSimpleSubalgebraTreeSU` to obtain information about irreducible simple subalgebras of $\mathfrak{su}(12)$: A_{11} ($\mathfrak{su}(12)$) is the root of the tree, C_6 corresponds to a proper subalgebra of A_{11}, and A_1 is a proper subalgebra of C_6. In addition, we have used `RepresentationDimension` to verify the dimensions of the representations. Let us use

`RepresentationDimension` to see what other $\mathfrak{su}(d)$ the Lie algebra of type C_6 should at the very least occur in:

```
> V := RSpace(Integers(), 6);
> [ RepresentationDimension(RootDatum("C6"), v) : v in Basis(V) ];
[ 12, 65, 208, 429, 572, 429 ]
```

We compare that to the list of $\mathfrak{su}(d)$ it does occur in using `IrreducibleSimpleSubalgebraTreeSU` and obtain the weights for the case $\mathfrak{su}(78)$.

```
> [ i : i in [2..2^10] | exists{r : r in VertexLabels(
>    IrreducibleSimpleSubalgebraTreeSU(Q, i)) | r`algebra eq "C6"} ];
[ 12, 65, 78, 208, 364, 429, 560, 572 ]
> t := IrreducibleSimpleSubalgebraTreeSU(Q, 78);
> l := VertexLabels(t);
> [ r`algebra : r in l ];
[ A77, C39, A1, D39, B6, C6, E6, A2, A11, A12 ]
> r := l[6];
> [ Matrix(x) : x in r`weights ];
[
    [2 0 0 0 0 0]
]
```

110.9 Bibliography

[**And77**] C.M. Andersen. Clebsch-Gordan Series for symmetrized tensor products. *J. Math. Phys*, 8:988–997, 1977.

[**CMT04**] Arjeh M. Cohen, Scott H. Murray, and D. E. Taylor. Computing in groups of Lie type. *Math. Comp.*, 73(247):1477–1498, 2004.

[**dG01**] W. A. de Graaf. Constructing representations of split semisimple Lie algebras. *J. Pure Appl. Algebra*, 164(1-2):87–107, 2001. Effective methods in algebraic geometry (Bath, 2000).

[**Dyn57**] E. B. Dynkin. Maximal Subgroups of the Classical Groups. *Amer. Math. Soc. Transl. Ser. 2*, 6:245–378, 1957.

[**FMS97**] Philippe Di Francesco, P. Mathieu, and D. Sénéchal. *Conformal Field Theory.* Graduate texts in contemporary physics. Springer, 1997.

[**JK81**] G. James and A. Kerber. *The Representation Theory of the Symmetric Group.* Addison-Weasly, Reading MA, 1981.

[**KL79**] D. Kazhdan and G. Lusztig. Representations of Coxeter groups and Hecke algebras. *Inventiones Math.*, 53:165–184, 1979.

[**vLCL92**] M.A.A. van Leeuwen, A.M. Cohen, and B. Lisser. *LiE, A package for Lie Group Computations.* CAN, Amsterdam, 1992.

[**ZSH11**] Robert Zeier and Thomas Schulte-Herbrüggen. Symmetry principles in quantum systems theory. *Journal of Mathematical Physics*, 52(11):113510, 2011.