

# HANDBOOK OF MAGMA FUNCTIONS

## **Volume 12**

### **Finite Geometry and Combinatorics**

John Cannon      Wieb Bosma

Claus Fieker      Allan Steel

Editors

Version 2.19

**Sydney**

April 24, 2013



# 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 Roozemon, 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 12: OVERVIEW

<b>XVIII</b>	<b>TOPOLOGY . . . . .</b>	<b>4689</b>
140	SIMPLICIAL HOMOLOGY	4691
<b>XIX</b>	<b>GEOMETRY . . . . .</b>	<b>4711</b>
141	FINITE PLANES	4713
142	INCIDENCE GEOMETRY	4749
143	CONVEX POLYTOPES AND POLYHEDRA	4771
<b>XX</b>	<b>COMBINATORICS . . . . .</b>	<b>4803</b>
144	ENUMERATIVE COMBINATORICS	4805
145	PARTITIONS, WORDS AND YOUNG TABLEAUX	4811
146	SYMMETRIC FUNCTIONS	4845
147	INCIDENCE STRUCTURES AND DESIGNS	4871
148	HADAMARD MATRICES	4907
149	GRAPHS	4917
150	MULTIGRAPHS	4999
151	NETWORKS	5047

# VOLUME 12: CONTENTS

<b>XVIII</b>	<b>TOPOLOGY</b>	<b>4689</b>
140	SIMPLICIAL HOMOLOGY . . . . .	4691
140.1	Introduction	4693
140.2	Simplicial Complexes	4693
140.2.1	Standard Topological Objects	4704
140.3	Homology Computation	4705
140.4	Bibliography	4709

<b>XIX</b>	<b>GEOMETRY</b>	<b>4711</b>
141	FINITE PLANES . . . . .	4713
141.1	Introduction	4715
141.1.1	Planes in Magma	4715
141.2	Construction of a Plane	4715
141.3	The Point-Set and Line-Set of a Plane	4718
141.3.1	Introduction	4718
141.3.2	Creating Point-Sets and Line-Sets	4718
141.3.3	Using the Point-Set and Line-Set to Create Points and Lines	4718
141.3.4	Retrieving the Plane from Points, Lines, Point-Sets and Line-Sets	4722
141.4	The Set of Points and Set of Lines	4722
141.5	The Defining Points of a Plane	4723
141.6	Subplanes	4724
141.7	Structures Associated with a Plane	4725
141.8	Numerical Invariants of a Plane	4726
141.9	Properties of Planes	4727
141.10	Identity and Isomorphism	4727
141.11	The Connection between Projective and Affine Planes	4728
141.12	Operations on Points and Lines	4729
141.12.1	Elementary Operations	4729
141.12.2	Deconstruction Functions	4730
141.12.3	Other Point and Line Functions	4733
141.13	Arcs	4734
141.14	Unitals	4737
141.15	The Collineation Group of a Plane	4738
141.15.1	The Collineation Group Function	4739
141.15.2	General Action of Collineations	4740
141.15.3	Central Collineations	4744
141.15.4	Transitivity Properties	4745
141.16	Translation Planes	4746
141.17	Planes and Designs	4746
141.18	Planes, Graphs and Codes	4747
142	INCIDENCE GEOMETRY . . . . .	4749
142.1	Introduction	4751
142.2	Construction of Incidence and Coset Geometries	4752
142.2.1	Construction of an Incidence Geometry	4752
142.2.2	Construction of a Coset Geometry	4756
142.3	Elementary Invariants	4759
142.4	Conversion Functions	4761
142.5	Residues	4762
142.6	Truncations	4763
142.7	Shadows	4763
142.8	Shadow Spaces	4763
142.9	Automorphism Group and Correlation Group	4764
142.10	Properties of Incidence Geometries and Coset Geometries	4764
142.11	Intersection Properties of Coset Geometries	4765
142.12	Primitivity Properties on Coset Geometries	4766
142.13	Diagram of an Incidence Geometry	4767
142.14	Bibliography	4770

143	CONVEX POLYTOPES AND POLYHEDRA . . . . .	4771
143.1	<i>Introduction and First Examples</i>	4773
143.2	<i>Polytopes, Cones and Polyhedra</i>	4778
143.2.1	Polytopes	4778
143.2.2	Cones	4779
143.2.3	Polyhedra	4780
143.2.4	Arithmetic Operations on Polyhedra	4782
143.3	<i>Basic Combinatorics of Polytopes and Polyhedra</i>	4783
143.3.1	Vertices and Inequalities	4783
143.3.2	Facets and Faces	4785
143.4	<i>The Combinatorics of Polytopes</i>	4786
143.4.1	Points in Polytopes	4786
143.4.2	Ehrhart Theory of Polytopes	4787
143.4.3	Automorphisms of a Polytope	4787
143.4.4	Operations on Polytopes	4788
143.5	<i>Cones and Polyhedra</i>	4788
143.5.1	Generators of Cones	4788
143.5.2	Properties of Polyhedra	4789
143.5.3	Attributes of Polyhedra	4793
143.5.4	Combinatorics of Polyhedral Complexes	4793
143.6	<i>Toric Lattices</i>	4793
143.6.1	Toric Lattices	4794
143.6.2	Points of Toric Lattices	4795
143.6.3	Operations on Toric Lattices	4798
143.6.4	Maps of Toric Lattices	4800
143.7	<i>Bibliography</i>	4801



<b>XX</b>	<b>COMBINATORICS</b>	<b>4803</b>
144	ENUMERATIVE COMBINATORICS . . . . .	4805
144.1	Introduction	4807
144.2	Combinatorial Functions	4807
144.3	Subsets of a Finite Set	4809
145	PARTITIONS, WORDS AND YOUNG TABLEAUX . . . . .	4811
145.1	Introduction	4813
145.2	Partitions	4813
145.3	Words	4816
145.3.1	Ordered Monoids	4816
145.3.2	Plactic Monoids	4819
145.4	Tableaux	4822
145.4.1	Tableau Monoids	4822
145.4.2	Creation of Tableaux	4824
145.4.3	Enumeration of Tableaux	4827
145.4.4	Random Tableaux	4829
145.4.5	Basic Access Functions	4830
145.4.6	Properties	4833
145.4.7	Operations	4835
145.4.8	The Robinson-Schensted-Knuth Correspondence	4838
145.4.9	Counting Tableaux	4842
145.5	Bibliography	4844
146	SYMMETRIC FUNCTIONS . . . . .	4845
146.1	Introduction	4847
146.2	Creation	4849
146.2.1	Creation of Symmetric Function Algebras	4849
146.2.2	Creation of Symmetric Functions	4851
146.3	Structure Operations	4854
146.3.1	Related Structures	4854
146.3.2	Ring Predicates and Booleans	4855
146.3.3	Predicates on Basis Types	4855
146.4	Element Operations	4855
146.4.1	Parent and Category	4855
146.4.2	Print Styles	4856
146.4.3	Additive Arithmetic Operators	4856
146.4.4	Multiplication	4857
146.4.5	Plethysm	4858
146.4.6	Boolean Operators	4858
146.4.7	Accessing Elements	4859
146.4.8	Multivariate Polynomials	4860
146.4.9	Frobenius Homomorphism	4861
146.4.10	Inner Product	4862
146.4.11	Combinatorial Objects	4862
146.4.12	Symmetric Group Character	4862
146.4.13	Restrictions	4863
146.5	Transition Matrices	4864
146.5.1	Transition Matrices from Schur Basis	4864
146.5.2	Transition Matrices from Monomial Basis	4866
146.5.3	Transition Matrices from Homogeneous Basis	4867
146.5.4	Transition Matrices from Power Sum Basis	4868

146.5.5	Transition Matrices from Elementary Basis	4869
146.6	<i>Bibliography</i>	4870
147	INCIDENCE STRUCTURES AND DESIGNS . . . . .	4871
147.1	<i>Introduction</i>	4873
147.2	<i>Construction of Incidence Structures and Designs</i>	4874
147.3	<i>The Point-Set and Block-Set of an Incidence Structure</i>	4878
147.3.1	Introduction	4878
147.3.2	Creating Point-Sets and Block-Sets	4879
147.3.3	Creating Points and Blocks	4879
147.4	<i>General Design Constructions</i>	4881
147.4.1	The Construction of Related Structures	4881
147.4.2	The Witt Designs	4884
147.4.3	Difference Sets and their Development	4884
147.5	<i>Elementary Invariants of an Incidence Structure</i>	4886
147.6	<i>Elementary Invariants of a Design</i>	4887
147.7	<i>Operations on Points and Blocks</i>	4889
147.8	<i>Elementary Properties of Incidence Structures and Designs</i>	4891
147.9	<i>Resolutions, Parallelisms and Parallel Classes</i>	4893
147.10	<i>Conversion Functions</i>	4896
147.11	<i>Identity and Isomorphism</i>	4897
147.12	<i>The Automorphism Group of an Incidence Structure</i>	4898
147.12.1	Construction of Automorphism Groups	4898
147.12.2	Action of Automorphisms	4901
147.13	<i>Incidence Structures, Graphs and Codes</i>	4903
147.14	<i>Automorphisms of Matrices</i>	4904
147.15	<i>Bibliography</i>	4905
148	HADAMARD MATRICES . . . . .	4907
148.1	<i>Introduction</i>	4909
148.2	<i>Equivalence Testing</i>	4909
148.3	<i>Associated 3-Designs</i>	4911
148.4	<i>Automorphism Group</i>	4912
148.5	<i>Databases</i>	4912
148.5.1	Updating the Databases	4913
149	GRAPHS . . . . .	4917
149.1	<i>Introduction</i>	4921
149.2	<i>Construction of Graphs and Digraphs</i>	4922
149.2.1	Bounds on the Graph Order	4922
149.2.2	Construction of a General Graph	4923
149.2.3	Construction of a General Digraph	4926
149.2.4	Operations on the Support	4928
149.2.5	Construction of a Standard Graph	4929
149.2.6	Construction of a Standard Digraph	4931
149.3	<i>Graphs with a Sparse Representation</i>	4932
149.4	<i>The Vertex-Set and Edge-Set of a Graph</i>	4934
149.4.1	Introduction	4934
149.4.2	Creating Edges and Vertices	4934
149.4.3	Operations on Vertex-Sets and Edge-Sets	4936
149.4.4	Operations on Edges and Vertices	4937
149.5	<i>Labelled, Capacitated and Weighted Graphs</i>	4938

149.6	<i>Standard Constructions for Graphs</i>	4938
149.6.1	Subgraphs and Quotient Graphs	4938
149.6.2	Incremental Construction of Graphs	4940
149.6.3	Constructing Complements, Line Graphs; Contraction, Switching	4943
149.7	<i>Unions and Products of Graphs</i>	4945
149.8	<i>Converting between Graphs and Digraphs</i>	4947
149.9	<i>Construction from Groups, Codes and Designs</i>	4947
149.9.1	Graphs Constructed from Groups	4947
149.9.2	Graphs Constructed from Designs	4948
149.9.3	Miscellaneous Graph Constructions	4949
149.10	<i>Elementary Invariants of a Graph</i>	4950
149.11	<i>Elementary Graph Predicates</i>	4951
149.12	<i>Adjacency and Degree</i>	4953
149.12.1	Adjacency and Degree Functions for a Graph	4953
149.12.2	Adjacency and Degree Functions for a Digraph	4954
149.13	<i>Connectedness</i>	4956
149.13.1	Connectedness in a Graph	4956
149.13.2	Connectedness in a Digraph	4957
149.13.3	Graph Triconnectivity	4957
149.13.4	Maximum Matching in Bipartite Graphs	4959
149.13.5	General Vertex and Edge Connectivity in Graphs and Digraphs	4960
149.14	<i>Distances, Paths and Circuits in a Graph</i>	4963
149.14.1	Distances, Paths and Circuits in a Possibly Weighted Graph	4963
149.14.2	Distances, Paths and Circuits in a Non-Weighted Graph	4963
149.15	<i>Maximum Flow, Minimum Cut, and Shortest Paths</i>	4964
149.16	<i>Matrices and Vector Spaces Associated with a Graph or Digraph</i>	4965
149.17	<i>Spanning Trees of a Graph or Digraph</i>	4965
149.18	<i>Directed Trees</i>	4966
149.19	<i>Colourings</i>	4967
149.20	<i>Cliques, Independent Sets</i>	4968
149.21	<i>Planar Graphs</i>	4973
149.22	<i>Automorphism Group of a Graph or Digraph</i>	4976
149.22.1	The Automorphism Group Function	4976
149.22.2	nauty Invariants	4977
149.22.3	Graph Colouring and Automorphism Group	4979
149.22.4	Variants of Automorphism Group	4980
149.22.5	Action of Automorphisms	4984
149.23	<i>Symmetry and Regularity Properties of Graphs</i>	4987
149.24	<i>Graph Databases and Graph Generation</i>	4989
149.24.1	Strongly Regular Graphs	4989
149.24.2	Small Graphs	4991
149.24.3	Generating Graphs	4992
149.24.4	A General Facility	4995
149.25	<i>Bibliography</i>	4997
150	MULTIGRAPHS . . . . .	4999
150.1	<i>Introduction</i>	5003
150.2	<i>Construction of Multigraphs</i>	5004
150.2.1	Construction of a General Multigraph	5004
150.2.2	Construction of a General Multidigraph	5005
150.2.3	Printing of a Multi(di)graph	5006
150.2.4	Operations on the Support	5007
150.3	<i>The Vertex-Set and Edge-Set of Multigraphs</i>	5008
150.4	<i>Vertex and Edge Decorations</i>	5011
150.4.1	Vertex Decorations: Labels	5011

150.4.2	Edge Decorations	5012
150.4.3	Unlabelled, or Uncapacitated, or Unweighted Graphs	5015
150.5	<i>Standard Construction for Multigraphs</i>	5018
150.5.1	Subgraphs	5018
150.5.2	Incremental Construction of Multigraphs	5020
150.5.3	Vertex Insertion, Contraction	5024
150.5.4	Unions of Multigraphs	5025
150.6	<i>Conversion Functions</i>	5026
150.6.1	Orientated Graphs	5027
150.6.2	Converse	5027
150.6.3	Converting between Simple Graphs and Multigraphs	5027
150.7	<i>Elementary Invariants and Predicates for Multigraphs</i>	5028
150.8	<i>Adjacency and Degree</i>	5030
150.8.1	Adjacency and Degree Functions for Multigraphs	5031
150.8.2	Adjacency and Degree Functions for Multidigraphs	5032
150.9	<i>Connectedness</i>	5033
150.9.1	Connectedness in a Multigraph	5034
150.9.2	Connectedness in a Multidigraph	5034
150.9.3	Triconnectivity for Multigraphs	5035
150.9.4	Maximum Matching in Bipartite Multigraphs	5035
150.9.5	General Vertex and Edge Connectivity in Multigraphs and Multidigraphs	5035
150.10	<i>Spanning Trees</i>	5037
150.11	<i>Planar Graphs</i>	5038
150.12	<i>Distances, Shortest Paths and Minimum Weight Trees</i>	5042
150.13	<i>Bibliography</i>	5046
151	NETWORKS . . . . .	5047
151.1	<i>Introduction</i>	5049
151.2	<i>Construction of Networks</i>	5049
151.2.1	Magma Output: Printing of a Network	5051
151.3	<i>Standard Construction for Networks</i>	5053
151.3.1	Subgraphs	5053
151.3.2	Incremental Construction: Adding Edges	5057
151.3.3	Union of Networks	5058
151.4	<i>Maximum Flow and Minimum Cut</i>	5059
151.5	<i>Bibliography</i>	5065

# PART XVIII

## TOPOLOGY

140      SIMPLICIAL HOMOLOGY

4691



# 140 SIMPLICIAL HOMOLOGY

<b>140.1 Introduction . . . . .</b>	<b>4693</b>	<b>Cone(X)</b>	<b>4703</b>
		<b>Suspension(X)</b>	<b>4703</b>
<b>140.2 Simplicial Complexes . . .</b>	<b>4693</b>	<i>140.2.1 Standard Topological Objects . .</i>	<i>4704</i>
<b>SimplicialComplex(f)</b>	<b>4693</b>	<b>Simplex(n)</b>	<b>4704</b>
<b>SimplicialComplex(G)</b>	<b>4694</b>	<b>Sphere(n)</b>	<b>4704</b>
<b>FlagComplex(G)</b>	<b>4694</b>	<b>KleinBottle()</b>	<b>4704</b>
<b>CliqueComplex(G)</b>	<b>4694</b>	<b>Torus()</b>	<b>4704</b>
<b>Dimension(X)</b>	<b>4694</b>	<b>Cylinder()</b>	<b>4704</b>
<b>Faces(X, d)</b>	<b>4695</b>	<b>MoebiusStrip()</b>	<b>4704</b>
<b>Facets(X)</b>	<b>4695</b>	<b>LensSpace(p)</b>	<b>4704</b>
<b>Normalization(X)</b>	<b>4696</b>	<b>SimplicialProjectivePlane()</b>	<b>4704</b>
<b>Normalization(~X)</b>	<b>4696</b>	<b>140.3 Homology Computation . .</b>	<b>4705</b>
<b>Shift(X, n)</b>	<b>4696</b>	<b>Homology(X)</b>	<b>4705</b>
<b>Shift(~X, n)</b>	<b>4696</b>	<b>Homology(~X)</b>	<b>4705</b>
<b>Boundary(X)</b>	<b>4697</b>	<b>Homology(X, A)</b>	<b>4705</b>
<b>+</b>	<b>4698</b>	<b>Homology(~X, A)</b>	<b>4705</b>
<b>eq</b>	<b>4698</b>	<b>HomologyGroup(X, q)</b>	<b>4706</b>
<b>Product(S, T)</b>	<b>4699</b>	<b>HomologyGroup(X, q, A)</b>	<b>4706</b>
<b>Join(S, T)</b>	<b>4700</b>	<b>BettiNumber(X, q)</b>	<b>4706</b>
<b>*</b>	<b>4700</b>	<b>BettiNumber(X, q, A)</b>	<b>4706</b>
<b>AddSimplex(X, s)</b>	<b>4701</b>	<b>TorsionCoefficients(X, q)</b>	<b>4706</b>
<b>AddSimplex(~X, s)</b>	<b>4701</b>	<b>TorsionCoefficients(X, q, A)</b>	<b>4706</b>
<b>AddSimplex(X, s)</b>	<b>4701</b>	<b>EulerCharacteristic(X)</b>	<b>4706</b>
<b>Addsimplex(~X, s)</b>	<b>4701</b>	<b>BoundaryMatrix(X, q, A)</b>	<b>4706</b>
<b>Prune(X, f)</b>	<b>4701</b>	<b>ChainComplex(X, A)</b>	<b>4707</b>
<b>Prune(~X, f)</b>	<b>4701</b>	<b>HomologyGenerators(X)</b>	<b>4707</b>
<b>Glue(X, e)</b>	<b>4701</b>	<b>HomologyGenerators(X, A)</b>	<b>4707</b>
<b>Glue(~X, e)</b>	<b>4701</b>	<b>HomologyGenerators(H, M, X)</b>	<b>4707</b>
<b>BarycentricSubdivision(X)</b>	<b>4702</b>	<b>140.4 Bibliography . . . . .</b>	<b>4709</b>
<b>Skeleton(X, q)</b>	<b>4703</b>		
<b>UnderlyingGraph(X)</b>	<b>4703</b>		





# Chapter 140

## SIMPLICIAL HOMOLOGY

### 140.1 Introduction

This chapter presents the category of finite simplicial complexes.

We define an abstract simplicial complex  $K$  to be a subset of the power set of some set  $V$  of vertices, with the property that if  $S \in K$  and  $T \subset S$  then  $T \in K$ .

For detailed reading on simplicial complexes and their homology, we refer to [Hat02] and [Arm83].

Simplicial complexes may be defined over any `SetEnum`, however, many of the construction methods operate over `SetEnum[RngIntElt]`. The handbook refers to such simplicial complexes as *normalized*.

A simplicial complex carries the category name `SmpCpx`. Constructors and package internal functions guarantee that the closure under subsets relation is kept intact.

### 140.2 Simplicial Complexes

The module supports creation of simplicial complexes from lists of faces, as well as a few preprogrammed complex types. Furthermore, several standard techniques for modifying and recombining simplicial complexes are available.

`SimplicialComplex(f)`

Constructs an abstract simplicial complex with the faces in the list  $f$ . The argument should be a sequence of sets. There is no requirement on the type of the elements in the face sets, however several of the constructions require the sets to have integer entries. See **Normalization** on page 4696 for automated renumbering of the face set elements.

#### Example H140E1

---

We give a simplicial complex by listing the facets, or also redundant faces if we want to.

```
> sc := SimplicialComplex([{1,2,3},{1,2,4},{1,3,4},{2,3,4},{1,5},{2,5}]);
```

We can view the facets (faces not included in any other faces) of the complex, or by giving the `:Maximal` output flag, we can view all faces.

```
> sc;
Simplicial complex
[
  { 2, 3, 4 },
  { 1, 2, 3 },
  { 1, 2, 4 },
```

```

    { 2, 5 },
    { 1, 5 },
    { 1, 3, 4 }
]
> sc:Maximal;
Simplicial complex
{
    {},
    { 4 },
    { 2, 3, 4 },
    { 2, 3 },
    { 3, 4 },
    { 1, 3 },
    { 3 },
    { 1 },
    { 1, 4 },
    { 1, 2, 3 },
    { 2 },
    { 1, 2 },
    { 1, 2, 4 },
    { 2, 5 },
    { 1, 5 },
    { 5 },
    { 2, 4 },
    { 1, 3, 4 }
}

```

---

SimplicialComplex(G)

Constructs a 1-dimensional simplicial complex isomorphic to the graph  $G$ .

FlagComplex(G)

CliqueComplex(G)

Constructs a simplicial complex with an  $n$ -simplex for each  $n$ -clique in the graph  $G$ .

Dimension(X)

Returns the dimension of the highest dimensional face of the simplicial complex  $X$ . Note that there is a difference between degrees and dimensions of faces. A face is said to have degree  $n$  if there are  $n$  elements in the face as a set, and it is said to have dimension  $n$  if there are  $n + 1$  elements in the face as a set.

**Example H140E2**

---

The recently defined simplicial complex has top dimension 2.

```
> Dimension(sc);  
2
```

---

**Faces( $X$ ,  $d$ )**

Returns the faces of one degree  $d$  of the simplicial complex  $X$ . Recall that the degree of a face is the number of elements in the face as a set, and one more than the dimension of the face.

The order of faces returned for each degree is also the correspondence between faces and the basis of the corresponding module in a chain complex constructed from the simplicial complex.

If the complex does not possess any faces of the requested degree, an empty sequence is returned.

**Example H140E3**

---

We can list faces of any degree in the defined simplicial complex.

```
> Faces(sc,2);  
[  
  { 2, 3 },  
  { 3, 4 },  
  { 1, 3 },  
  { 1, 4 },  
  { 1, 2 },  
  { 2, 5 },  
  { 1, 5 },  
  { 2, 4 }  
]  
> Faces(sc,5);  
[]
```

---

**Facets( $X$ )**

Returns the facets of the simplicial complex  $X$ . The facets of a simplicial complex are the faces that are not themselves subsets of another face. This is what the normal printing mode for a simplicial complex outputs.

**Example H140E4**

---

We can read the facets both by fetching the sequence of facets and by printing the complex as such.

```
> Facets(sc);
[
  { 2, 3, 4 },
  { 1, 2, 3 },
  { 1, 2, 4 },
  { 2, 5 },
  { 1, 5 },
  { 1, 3, 4 }
]
> sc;
Simplicial complex
[
  { 2, 3, 4 },
  { 1, 2, 3 },
  { 1, 2, 4 },
  { 2, 5 },
  { 1, 5 },
  { 1, 3, 4 }
]
```

---

Normalization(X)
------------------

Normalization( $\sim$ X)
--------------------------

Relabels the points (elements of the face sets) of the simplicial complex  $X$  using  $1, 2, 3, \dots$ . This ensures that the simplicial complex is built on subsets of the integers, which is a prerequisite for several functions handling simplicial complexes. We call a simplicial complex on subsets of the integers *normalized* in this handbook.

Note the two calling signatures of this function. The first signature copies the simplicial complex and returns a new complex with the desired modification, whereas the second signature modifies the complex in place.

Shift(X, n)
-------------

Shift( $\sim$ X, n)
---------------------

Shifts all integer points in the normalized simplicial complex  $X$  by an offset given by  $n$ .

Note the two calling signatures of this function. The first signature copies the simplicial complex and returns a new complex with the desired modification, whereas the second signature modifies the complex in place.

**Example H140E5**

---

We define a simplicial complex using string labels, normalize it and then shift the labels.

```
> cpx := Boundary(SimplicialComplex(["a","b","c"]));
> cpx;
Simplicial complex
[
  { c, a },
  { c, b },
  { b, a }
]
> Normalization(~cpx);
> cpx;
Simplicial complex
[
  { 1, 3 },
  { 2, 3 },
  { 1, 2 }
]
> Shift(~cpx,-2);
> cpx;
Simplicial complex
[
  { -1, 1 },
  { 0, 1 },
  { -1, 0 }
]
```

Boundary( $X$ )

Returns the simplicial complex generated by all simplexes that lie in the boundary of the simplicial complex  $X$ . This can be used to easily acquire a simplicial complex representing the  $n$ -sphere.

**Example H140E6**

---

We can determine the boundary of our previously defined simplicial complex, or construct a 4-sphere.

```
> Boundary(sc);
Simplicial complex
[
  { 2, 3 },
  { 3, 4 },
  { 1, 3 },
  { 1, 4 },
  { 1, 2 },
```

```

    { 5 },
    { 2, 4 }
]
> sph4 := Boundary(SimplicialComplex([{1,2,3,4,5,6}]));
> sph4;
Simplicial complex
[
  { 1, 2, 3, 5, 6 },
  { 2, 3, 4, 5, 6 },
  { 1, 3, 4, 5, 6 },
  { 1, 2, 3, 4, 5 },
  { 1, 2, 4, 5, 6 },
  { 1, 2, 3, 4, 6 }
]
```

---

S + T

Returns the topological sum, or disjoint union, of two simplicial complexes. Requires both complexes to be normalized using [Normalization](#) on page 4696.

### Example H140E7

---

We can construct a disjoint union of two circles.

```

> sph2 := Boundary(SimplicialComplex([{1,2,3}]));
> sph2 + sph2;
Simplicial complex
[
  { 4, 6 },
  { 2, 3 },
  { 4, 5 },
  { 5, 6 },
  { 1, 3 },
  { 1, 2 }
]
```

---

S eq T

Compares two simplicial complexes. Will not try to find an isomorphism, rather does the comparison by ensuring that the points are in the same universe and then doing a set comparison on the set of faces.

**Example H140E8**

---

Shifting a complex, for instance, will break equality, since the labels differ.

```
> sc eq Shift(sc,2);
false
> sc eq Shift(Shift(sc,2),-2);
true
```

Furthermore, isomorphic complexes with different labels will not be equal.

```
> circ1 := Boundary(SimplicialComplex([1,2,3]));
> circ2 := Boundary(SimplicialComplex(["a","b","c"]));
> circ1 eq circ2;
false
```

---

## Product(S,T)

Returns the cartesian product of two simplicial complexes. This will work on any complexes, since the new points will be pairs of points from the component complexes.

**Example H140E9**

---

Using the two different circles from the last example, we can now construct a torus.

```
> Product(circ1,circ2);
Simplicial complex
[
  { <3, b>, <1, b>, <3, c> },
  { <1, c>, <3, c>, <1, a> },
  { <2, a>, <3, b>, <3, a> },
  { <3, b>, <1, b>, <1, a> },
  { <1, b>, <2, b>, <2, c> },
  { <2, a>, <3, c>, <3, a> },
  { <1, c>, <1, b>, <3, c> },
  { <3, c>, <2, b>, <2, c> },
  { <1, c>, <1, b>, <2, c> },
  { <3, c>, <3, a>, <1, a> },
  { <2, a>, <3, c>, <2, c> },
  { <3, b>, <3, a>, <1, a> },
  { <2, a>, <1, a>, <2, b> },
  { <2, a>, <1, a>, <2, c> },
  { <1, c>, <1, a>, <2, c> },
  { <3, b>, <3, c>, <2, b> },
  { <1, b>, <1, a>, <2, b> },
  { <2, a>, <3, b>, <2, b> }
```

]

If we want something less unwieldy – especially after repeated cartesian products – we can always normalize the resulting complexes.

```
> line := SimplicialComplex([{1,2}]);
> square := Product(line,line);
> cube1 := Product(square,line);
> cube2 := Product(line,square);
> cube1 eq cube2;
false
> Normalization(cube1) eq Normalization(cube2);
false
> cube1;
Simplicial complex
[
  { <<1, 2>, 1>, <<1, 2>, 2>, <<1, 1>, 1>, <<2, 2>, 2> },
  { <<2, 1>, 1>, <<2, 2>, 1>, <<1, 1>, 1>, <<2, 2>, 2> },
  { <<1, 2>, 2>, <<1, 1>, 1>, <<1, 1>, 2>, <<2, 2>, 2> },
  { <<1, 2>, 1>, <<2, 2>, 1>, <<1, 1>, 1>, <<2, 2>, 2> },
  { <<1, 1>, 1>, <<1, 1>, 2>, <<2, 1>, 2>, <<2, 2>, 2> },
  { <<2, 1>, 1>, <<1, 1>, 1>, <<2, 1>, 2>, <<2, 2>, 2> }
]
> Normalization(cube1);
Simplicial complex
[
  { 3, 4, 5, 8 },
  { 4, 5, 6, 8 },
  { 1, 3, 4, 5 },
  { 2, 4, 5, 7 },
  { 1, 2, 4, 5 },
  { 4, 5, 6, 7 }
]
```

Join(S,T)
-----------

S * T
-------

Constructs the join of two simplicial complexes. The join of two simplicial complexes is defined as the complex generated by faces on the form  $\{x_1, \dots, x_r, y_1, \dots, y_s\}$  for  $\{x_1, \dots, x_r\} \in S$  and  $\{y_1, \dots, y_s\} \in T$ . Requires both simplicial complexes to be normalized.



**Example H140E10**

---

The join of two edges is a 3-simplex.

```
> SimplicialComplex([1,2]) * SimplicialComplex([1,2]);
Simplicial complex
[
  { 1, 2, 3, 4 }
]
```

---

AddSimplex( $X$ , $s$ )
-------------------------

AddSimplex( $\sim X$ , $s$ )
------------------------------

AddSimplex( $X$ , $s$ )
-------------------------

Addsimplex( $\sim X$ , $s$ )
------------------------------

Adds either a single simplex  $s$  expressed as a set or a sequence of simplexes to a simplicial complex. The functional versions of the function call return a new complex with the added simplices included, and the procedural change the complex in place.

Prune( $X$ , $f$ )
--------------------

Prune( $\sim X$ , $f$ )
-------------------------

Removes a face  $f$  and all faces containing  $f$  from a simplicial complex  $X$ . If  $f$  is not a face of  $X$ , then  $X$  is returned unchanged.

Note the two calling signatures of this function. The first signature copies the simplicial complex and returns a new complex with the desired modification, whereas the second signature modifies the complex in place.

Glue( $X$ , $e$ )
-------------------

Glue( $\sim X$ , $e$ )
------------------------

In the simplicial complex  $X$ , identify all points identified by pairs in the sequence  $e$ .

The sequence  $e$  should be a sequence of pairs of elements of the face sets of  $X$ . The identification will eliminate the first component of each pair, replacing it with the second component wherever it occurs in  $X$ .

Note the two calling signatures of this function. The first signature copies the simplicial complex and returns a new complex with the desired modification, whereas the second signature modifies the complex in place.

**Example H140E11**

---

We can, for instance, construct the connected sum of two tori using `Product`, `Prune`, the topological sum and `Glue`.

```
> circ := SimplicialComplex([{1,2},{2,3},{3,1}]);
> torus := Product(circ,circ);
> Normalization(~torus);
> torus;
Simplicial complex
[
  { 2, 3, 6 },
  { 1, 2, 9 },
  { 2, 6, 8 },
  { 4, 7, 8 },
  { 3, 5, 9 },
  { 4, 6, 7 },
  { 1, 8, 9 },
  { 1, 4, 8 },
  { 6, 8, 9 },
  { 2, 7, 8 },
  { 2, 3, 9 },
  { 5, 6, 9 },
  { 1, 3, 5 },
  { 1, 4, 6 },
  { 3, 6, 7 },
  { 1, 5, 6 },
  { 1, 2, 7 },
  { 1, 3, 7 }
]
> oneholetorus := Prune(torus,{1,2,7});
> twoholetorus := Prune(Prune(torus,{1,2,7}},{6,8,9});
> threetori := oneholetorus + twoholetorus + oneholetorus;
> threetorus := Glue(threetori,[<1,10>,<2,11>,<7,16>,<15,19>,<17,20>,<18,25>]);
```

Note that we find the glue data by considering that the maximal point in the original torus had number 9, so for the second added torus, all points will be shifted by 9 and in the third all points will be shifted by 18. Thus, the excised facets  $\{1, 2, 7\}$  and  $\{6, 8, 9\}$  turn into  $\{10, 11, 16\}$  and  $\{15, 17, 18\}$ , and the excised facet  $\{1, 2, 7\}$  in the third torus turns into  $\{19, 20, 25\}$ .

**BarycentricSubdivision(X)**

Constructs the barycentric subdivision of the simplicial complex  $X$ . Abstractly, this is a simplicial complex whose faces are chains  $X_1 \subset \dots \subset X_n$  of faces from  $X$ . The new complex has more faces but the same homotopy type as the old complex.

**Skeleton( $X$ ,  $q$ )**

Returns the  $q$ -skeleton of the simplicial complex  $X$ . This is the complex consisting of all faces of  $X$  of dimension at most  $q$ .

**UnderlyingGraph( $X$ )**

Constructs a graph isomorphic, as a graph, to the 1-skeleton of the simplicial complex  $X$ .

**Cone( $X$ )**

Constructs a cone over the simplicial complex  $X$ . The cone is generated by all faces of the complex, with an additional vertex included into each face. Any cone is acyclic, in other words all homology groups vanish. Requires a normalized simplicial complex.

**Suspension( $X$ )**

Constructs the suspension, or double cone, over the normalized simplicial complex  $X$ . The suspension has the added property that all the homology groups occur in it, shifted up by one dimension.

**Example H140E12**

---

For computation of the relevant homology, and a demonstration of the stated facts about the cone and suspension operations, please refer to examples in the Section on homology computation.

```
> circ := Boundary(SimplicialComplex([{1,2,3}]));
> Cone(circ);
Simplicial complex
[
  { 1, 3, 4 },
  { 2, 3, 4 },
  { 1, 2, 4 }
]
> Suspension(circ);
Simplicial complex
[
  { 1, 3, 5 },
  { 1, 3, 4 },
  { 1, 2, 5 },
  { 2, 3, 4 },
  { 1, 2, 4 },
  { 2, 3, 5 }
]
```

---

### 140.2.1 Standard Topological Objects

**Simplex(n)**

Returns the  $n$ -dimensional simplex as a simplicial complex.

**Sphere(n)**

Returns a simplicial complex triangulating the  $n$ -sphere.

**KleinBottle()**

Returns a triangulation of the Klein bottle as an abstract simplicial complex.

**Torus()**

Returns a triangulation of a torus as an abstract simplicial complex.

**Cylinder()**

Returns a triangulation of a cylinder as an abstract simplicial complex.

**MoebiusStrip()**

Returns a triangulation of the Moebius strip as an abstract simplicial complex.

**LensSpace(p)**

Constructs the lens space  $L(p, 1)$ . This has a 1-dimensional homology generator of order  $p$ .

**SimplicialProjectivePlane()**

Constructs a triangulation of the projective plane.

Examples for the usage of **LensSpace** and **SimplicialProjectivePlane** will be given in Section 140.3 on homology computation.

### 140.3 Homology Computation

The code computes exclusively reduced homology of the given simplicial complexes. If you want the non-reduced homology, just add a single free rank to dimension 0 and let it be generated by any single point in the complex.

Homology(X)
-------------

Homology( $\sim$ X)
---------------------

Homology(X,A)
---------------

Homology( $\sim$ X, A)
------------------------

Calculates the reduced homology of a simplicial complex  $X$  with coefficients in the ring  $A$ . The procedural form of this command caches the results of the calculation in the simplicial complex object. If no ring is given, then the function defaults to integer coefficients.

#### Example H140E13

---

The resulting modules are stored in falling dimension, always including the dimension  $-1$  vanishing homology module at the very end.

```
> circ := Boundary(SimplicialComplex([{1,2,3}]));
> Homology(circ,Integer());
[
  Full Quotient RSpace of degree 1 over Integer Ring
  Column moduli:
  [ 0 ],
  Full Quotient RSpace of degree 0 over Integer Ring
  Column moduli:
  [ ],
  Full Quotient RSpace of degree 0 over Integer Ring
  Column moduli:
  [ ]
]
[
  Mapping from: RSpace of degree 3, dimension 1 over Integer Ring to Full
  Quotient RSpace of degree 1 over Integer Ring
  Column moduli:
  [ 0 ],
  Mapping from: RSpace of degree 3, dimension 2 over Integer Ring to Full
  Quotient RSpace of degree 0 over Integer Ring
  Column moduli:
  [ ],
  Mapping from: Full RSpace of degree 1 over Integer Ring to Full Quotient
  RSpace of degree 0 over Integer Ring
  Column moduli:
  [ ]
]
```

```

> lens3 := LensSpace(3);
> Homology(~lens3,Integers());
> Homology(lens3,Integers())[3];
Full Quotient RSpace of degree 1 over Integer Ring
Column moduli:
[ 3 ]

```

---

HomologyGroup(X, q)
---------------------

HomologyGroup(X, q, A)
------------------------

Calculates and returns the  $q$ th homology group of  $X$  with coefficients in  $A$ . If no ring is given, the function defaults to integer coefficients. If the homology is cached in  $X$ , the cached results are returned. This function will not compute the entire homology in order to return one homology group.

BettiNumber(X,q)
------------------

BettiNumber(X,q,A)
--------------------

Returns the  $q$ th Betti number, computed as the free rank of the  $q$ -dimensional homology group, with coefficients in  $A$ . If no ring is given, then the function will default to integer coefficients.

Note that the Betti number computations compensate for the homology computations being reduced. Thus,  $\text{BettiNumber}(X,0) \text{ eq Rank}(\text{HomologyGroup}(X,0)) + 1$ .

TorsionCoefficients(X, q)
---------------------------

TorsionCoefficients(X, q, A)
------------------------------

Returns the torsion coefficients of the  $q$ th homology group of  $X$  with coefficients in  $A$ . If no ring is given, then the function will default to integer coefficients.

EulerCharacteristic(X)
------------------------

Computes the Euler characteristic of the complex. If homology is cached, this is used for computation, and else the characteristic is computed using the ranks of the chain groups.

BoundaryMatrix(X, q, A)
-------------------------

Returns the  $q$ th boundary matrix of the corresponding chain complex to the simplicial complex  $X$  with coefficients in  $A$ .

**ChainComplex(X, A)**

Constructs a reduced chain complex of free  $A$ -modules corresponding to the abstract simplicial complex  $X$ .

Note that the produced complex includes one extra rank 1 module on each end, with the zero map leading to it, to simulate the maps to and from the zero module that would end a chain complex constructed from a simplicial complex in ordinary cases.

**Example H140E14**

---

```
> ChainComplex(SimplicialComplex([1]),Integers());
Chain complex with terms of degree 2 down to -1
Dimensions of terms: 1 1 1 1
> BoundaryMaps(ChainComplex(SimplicialComplex([1]),Integers()));
[*
  [0],
  [-1],
  [0]
*]
> ChainComplex(SimplicialComplex([1,2,3]),GF(3));
Chain complex with terms of degree 4 down to -1
Dimensions of terms: 1 1 3 3 1 1
> BoundaryMaps(ChainComplex(SimplicialComplex([1,2,3]),GF(3)));
[*
  [0],
  [1 2 2],
  [1 0 2],
  [0 1 2],
  [1 2 0],
  [2],
  [2],
  [2],
  [0]
*]
```

**HomologyGenerators(X)****HomologyGenerators(X, A)****HomologyGenerators(H, M, X)**

Prints generators of the homology groups of the simplicial complex  $X$  with coefficients in  $A$  together with their order, in order of dimension. The latter calling form expects  $H, M$  to be the result from  $H, M := \text{ChainComplex}(A, \text{cmp})$ . This function will recalculate homology each time unless the homology is already cached in the simplicial complex using `Homology(A, cmp)`.

If no ring is given, the function defaults to integer coefficients.

**Example H140E15**

This function gives a condensed form of the actual bases of the homology groups, as well as mappings back to an actual chain representative for each homology class.

```
> HomologyGenerators(threetorus,Integers());
*** dimension 2 ***
inf: { 21, 23, 27 } - { 4, 6, 16 } - { 20, 24, 26 } + { 20, 25, 26 } -
{ 14, 19, 25 } + { 12, 14, 25 } + { 3, 6, 11 } - { 10, 14, 19 } +
{ 19, 20, 27 } - { 3, 9, 11 } - { 10, 13, 20 } + { 10, 13, 19 } -
{ 10, 12, 14 } - { 12, 16, 19 } - { 11, 19, 20 } - { 13, 16, 20 } +
{ 21, 24, 25 } + { 24, 26, 27 } - { 20, 21, 24 } - { 19, 22, 26 } -
{ 3, 10, 16 } - { 4, 6, 10 } - { 22, 25, 26 } + { 11, 16, 20 } +
{ 4, 8, 16 } - { 5, 6, 9 } + { 4, 8, 10 } - { 11, 12, 19 } - { 3, 5, 9 } +
{ 11, 12, 25 } + { 8, 9, 10 } + { 20, 21, 27 } + { 3, 6, 16 } +
{ 10, 11, 25 } + { 19, 22, 24 } - { 9, 10, 11 } + { 3, 5, 10 } +
{ 10, 20, 25 } - { 8, 11, 16 } - { 19, 26, 27 } - { 19, 23, 24 } +
{ 10, 12, 16 } - { 19, 21, 23 } - { 6, 8, 9 } + { 13, 16, 19 } +
{ 19, 21, 25 } + { 23, 24, 27 } - { 22, 24, 25 } + { 5, 6, 10 } +
{ 6, 8, 11 }
*** dimension 1 ***
inf: -1*{ 10, 13 } + { 3, 5 } - { 8, 9 } - { 8, 16 } - { 19, 22 } +
{ 20, 26 } + { 9, 11 } - { 3, 16 } + { 5, 10 } + { 22, 26 } + { 11, 20 } +
2*{ 10, 20 } + { 13, 19 } - 2*{ 10, 11 }
inf: { 3, 9 } - { 3, 5 } - { 9, 11 } - { 5, 10 } + { 10, 11 }
inf: -1*{ 26, 27 } - { 11, 19 } + 2*{ 19, 22 } - { 20, 26 } -
2*{ 22, 26 } + { 11, 20 } + { 19, 21 } - { 21, 27 }
inf: { 10, 13 } + { 11, 19 } - { 11, 20 } - { 10, 20 } - { 13, 19 }
inf: -1*{ 11, 19 } + { 19, 22 } - { 20, 26 } - { 22, 26 } + { 11, 20 }
inf: { 11, 20 } + { 10, 20 } - { 10, 11 }
```

The six found generators are the generators of each of the contained torus homology groups. Notice that each generator is printed out with a prefix. This gives the order of the generator - so that for instance torsion elements of homology may be recognized. Thus, we see with the projective plane:

```
> HomologyGenerators(SimplicialProjectivePlane(),Integers());
*** dimension 1 ***
2: { 3, 6 } + { 2, 3 } - { 2, 6 }
```

We can further take this opportunity to verify the claims about **Cone** and **Suspension** with regard to the homology.

```
> HomologyGenerators(Cone(SimplicialProjectivePlane()),Integers());
Complex is acyclic.
> HomologyGenerators(Suspension(SimplicialProjectivePlane()),Integers());
*** dimension 2 ***
2: { 1, 5, 7 } - { 1, 4, 7 } + { 1, 2, 5 } + { 1, 5, 8 } - { 2, 3, 8 } -
{ 3, 5, 7 } - { 1, 2, 8 } + { 5, 6, 8 } + { 1, 4, 5 } - { 4, 6, 8 } +
{ 3, 4, 7 } + { 2, 3, 5 } - { 4, 5, 6 } - { 3, 4, 8 }
```



## 140.4 Bibliography

- [**Arm83**] Mark Anthony Armstrong. *Basic topology*. Undergraduate Texts in Mathematics. Springer-Verlag, New York, 1983. Corrected reprint of the 1979 original.
- [**Hat02**] Allen Hatcher. *Algebraic topology*. Cambridge University Press, Cambridge, 2002.



# PART XIX

## GEOMETRY

141	FINITE PLANES	4713
142	INCIDENCE GEOMETRY	4749
143	CONVEX POLYTOPES AND POLYHEDRA	4771



# 141 FINITE PLANES

<b>141.1 Introduction . . . . .</b>	<b>4715</b>	<b>Support(P, p)</b>	<b>4723</b>
141.1.1 Planes in Magma . . . . .	4715	Support(p)	4723
<b>141.2 Construction of a Plane . .</b>	<b>4715</b>	<b>141.6 Subplanes . . . . .</b>	<b>4724</b>
FiniteProjectivePlane< >	4715	sub< >	4724
FiniteProjectivePlane< >	4715	SubfieldSubplane(P, F)	4724
FiniteProjectivePlane(W)	4716	<b>141.7 Structures Associated with a</b>	
FiniteProjectivePlane(F)	4716	<b>Plane . . . . .</b>	<b>4725</b>
FiniteProjectivePlane(q)	4716	VectorSpace(P)	4725
FiniteAffinePlane< >	4716	Field(P)	4725
FiniteAffinePlane< >	4716	IncidenceMatrix(P)	4725
FiniteAffinePlane(W)	4717	Dual(P)	4725
FiniteAffinePlane(F)	4717	<b>141.8 Numerical Invariants of a</b>	
FiniteAffinePlane(q)	4717	<b>Plane . . . . .</b>	<b>4726</b>
<b>141.3 The Point-Set and Line-Set of</b>		Order(P)	4726
<b>a Plane . . . . .</b>	<b>4718</b>	NumberOfPoints(P)	4726
141.3.1 Introduction . . . . .	4718	#	4726
141.3.2 Creating Point-Sets and Line-Sets	4718	NumberOfLines(P)	4726
PointSet(P)	4718	#	4726
LineSet(P)	4718	pRank(P)	4726
141.3.3 Using the Point-Set and Line-Set to		pRank(P, p)	4726
Create Points and Lines . . . .	4718	<b>141.9 Properties of Planes . . . .</b>	<b>4727</b>
.	4718	IsDesarguesian(P)	4727
!	4718	IsSelfDual(P)	4727
!	4718	<b>141.10 Identity and Isomorphism .</b>	<b>4727</b>
!	4719	eq	4727
Representative(V)	4719	ne	4727
Rep(V)	4719	IsIsomorphic(P, Q: -)	4727
Random(V)	4719	subset	4728
.	4719	<b>141.11 The Connection between Pro-</b>	
!	4719	<b>jective and Affine Planes . .</b>	<b>4728</b>
!	4719	FiniteAffinePlane(P, l)	4728
!	4719	ProjectiveEmbedding(P)	4728
Representative(L)	4719	<b>141.12 Operations on Points and</b>	
Rep(L)	4719	<b>Lines . . . . .</b>	<b>4729</b>
Random(L)	4719	141.12.1 Elementary Operations . . . .	4729
141.3.4 Retrieving the Plane from Points,		eq	4729
Lines, Point-Sets and Line-Sets .	4722	ne	4729
ParentPlane(V)	4722	eq	4730
ParentPlane(L)	4722	ne	4730
ParentPlane(p)	4722	in	4730
ParentPlane(l)	4722	notin	4730
<b>141.4 The Set of Points and Set of</b>		subset	4730
<b>Lines . . . . .</b>	<b>4722</b>	notsubset	4730
Points(P)	4722	meet	4730
Lines(P)	4722	Representative(l)	4730
<b>141.5 The Defining Points of a Plane</b>	<b>4723</b>	Rep(l)	4730
Support(P)	4723	Random(l)	4730
Support(l)	4723	141.12.2 Deconstruction Functions . . . .	4730

Index(P, p)	4730	141.15.1 The Collineation Group Function	4739
Index(P, l)	4730	CollineationGroup(P)	4739
p[i]	4730	AutomorphismGroup(P)	4739
l[i]	4731	PointGroup(P)	4739
Coordinates(P, p)	4731	LineGroup(P)	4739
Coordinates(P, l)	4731	CollineationGroupStabilizer(P, k)	4739
ElementToSequence(p)	4731	CollineationSubgroup(P)	4739
Eltseq(p)	4731	141.15.2 General Action of Collineations	4740
ElementToSequence(l)	4731	~	4740
Eltseq(l)	4731	~	4740
Set(l)	4731	Image(g, Y, y)	4740
141.12.3 Other Point and Line Functions	4733	Orbit(G, Y, y)	4740
IsCollinear(P, S)	4733	Orbits(G, Y)	4740
IsConcurrent(P, R)	4733	Stabilizer(G, Y, y)	4740
ContainsQuadrangle(P, S)	4733	Action(G, Y)	4741
Pencil(P, p)	4733	ActionImage(G, Y)	4741
Slope(l)	4733	ActionKernel(G, Y)	4741
IsParallel(P, l, m)	4733	141.15.3 Central Collineations	4744
ParallelClass(P, l)	4733	CentralCollineationGroup(P, p, l)	4744
ParallelClasses(P)	4733	CentralCollineationGroup(P, p)	4744
141.13 Arcs	4734	CentralCollineationGroup(P, l)	4744
kArc(P, k)	4734	IsCentralCollineation(P, g)	4744
CompleteKArc(P, k)	4734	141.15.4 Transitivity Properties	4745
IsArc(P, A)	4734	IsPointTransitive(P)	4745
IsComplete(P, A)	4734	IsTransitive(P)	4745
Conic(P, S)	4734	IsLineTransitive(P)	4745
QuadraticForm(S)	4735	141.16 Translation Planes	4746
Tangent(P, A, p)	4735	BaerDerivation(q2)	4746
AllTangents(P, A)	4735	BaerSubplane(P)	4746
AllSecants(P, A)	4735	OvalDerivation(q: -)	4746
ExternalLines(P, A)	4735	141.17 Planes and Designs	4746
AllPassants(P, A)	4735	Design(P)	4746
Knot(P, C)	4735	FiniteAffinePlane(D)	4746
Exterior(P, C)	4735	FiniteProjectivePlane(D)	4746
Interior(P, C)	4735	141.18 Planes, Graphs and Codes	4747
141.14 Unitals	4737	LineGraph(P)	4747
IsUnital(P, U)	4737	IncidenceGraph(P)	4747
AllTangents(P, U)	4737	LinearCode(P, K)	4748
UnitalFeet(P, U, p)	4737		
141.15 The Collineation Group of a Plane	4738		

# Chapter 141

## FINITE PLANES

### 141.1 Introduction

In this chapter we will present the categories of finite projective and affine planes.

The category names for projective and affine planes are `PlaneProj` and `PlaneAff` respectively. Within each of these categories we have what we will call *classical* planes — those which are defined by a vector space of dimension 2 (for affine planes) or 3 (for projective planes).

Some functions documented here apply to all types of planes, others are specific to projective, affine or classical planes. It should be clear which is the case for each entry.

#### 141.1.1 Planes in Magma

A point of a plane is considered to be a special object, and so points are given their own special type, `PlanePt`, in MAGMA. This allows the points of a plane to be defined over any type of MAGMA object, and also improves the efficiency of the code.

A special structure called the *point-set* acts as the parent structure for points. A point is created by coercing an appropriate MAGMA object into the point-set. It is also possible to get the  $i$ -th point, or a random point, from the point-set.

Similarly, lines of a plane have a special type `PlaneLn`, and the *line-set* acts as their parent structure. Lines can be created by coercing a suitable object into the line-set, or by asking for the  $i$ -th line, or a random line, from the line-set.

### 141.2 Construction of a Plane

All functions which create a plane return three values:

- (i) the plane itself;
- (ii) the point-set of the plane;
- (iii) the line-set of the plane.

These “sets” ((ii) and (iii)) are used as the parent structures for points and lines respectively, and are explained more fully in the next section.

<code>FiniteProjectivePlane&lt; v   X : parameters &gt;</code>
<code>FiniteProjectivePlane&lt; V   X : parameters &gt;</code>

Check

BOOLELT

Default : true

Construct the projective plane  $P$  having as point set the indexed set  $V$  (or  $\{@1, 2, \dots, v@\}$  if an integer  $v$  is given), and as line set  $L = \{L_1, L_2, \dots, L_b\}$  given by the list  $X$ . The value of  $X$  must be either:

- (a) A list of subsets of the set  $V$ .
- (b) A sequence, set or indexed set of subsets of  $V$ .
- (c) A list of lines of an existing plane.
- (d) A sequence, set or indexed set of lines of an existing plane.
- (e) A combination of the above.
- (f) A  $v \times b$   $(0, 1)$ -matrix  $A$ , where  $A$  may be defined over any coefficient ring. The matrix  $A$  will be interpreted as the incidence matrix for the plane  $P$ .
- (g) A set of codewords of a linear code with length  $v$ . The line set of  $P$  is taken to be the set of supports of the codewords.

The optional boolean argument **Check** indicates whether or not to check that the given data satisfies the projective plane axioms.

<b>FiniteProjectivePlane</b> ( $W$ )
--------------------------------------

<b>FiniteProjectivePlane</b> ( $F$ )
--------------------------------------

<b>FiniteProjectivePlane</b> ( $q$ )
--------------------------------------

Given a 3-dimensional vector space  $W$  defined over the field  $F = \mathbf{F}_q$ , construct the classical projective plane defined by the one-dimensional and two-dimensional subspaces of  $W$ .

<b>FiniteAffinePlane</b> < $v$   $X$ : <i>parameters</i> >
--

<b>FiniteAffinePlane</b> < $V$   $X$ : <i>parameters</i> >
--

**Check**

BOOLELT

*Default* : true

Construct the affine plane  $P$  having as point set the indexed set  $V$  (or  $\{ @1, 2, \dots, v@ \}$  if an integer  $v$  is given), and as line set  $L = \{ L_1, L_2, \dots, L_b \}$  given by the list  $X$ . The value of  $X$  must be either:

- (a) A list of subsets of the set  $V$ .
- (b) A sequence, set or indexed set of subsets of  $V$ .
- (c) A list of lines of an existing plane.
- (d) A sequence, set or indexed set of lines of an existing plane.
- (e) A combination of the above.
- (f) A  $v \times b$   $(0, 1)$ -matrix  $A$ , where  $A$  may be defined over any coefficient ring. The matrix  $A$  will be interpreted as the incidence matrix for the plane  $P$ .
- (g) A set of codewords of a linear code with length  $v$ . The line set of  $P$  is taken to be the set of supports of the codewords.

The optional boolean argument **Check** indicates whether or not to check that the given data satisfies the affine plane axioms.



FiniteAffinePlane(W)
----------------------

FiniteAffinePlane(F)
----------------------

FiniteAffinePlane(q)
----------------------

Given a 2-dimensional vector space  $W$  defined over the field  $F = \mathbf{F}_q$ , construct the classical affine plane defined by the cosets of the subspaces of  $W$ .

---

**Example H141E1**

The classical projective plane of order 3 can be constructed by the following statement:

```
> P, V, L := FiniteProjectivePlane(3);
> P;
Projective Plane PG(2, 3)
> V;
Point-set of Projective Plane PG(2, 3)
> L;
Line-set of Projective Plane PG(2, 3)
```

A non-classical affine plane of order 2 can be constructed in the following way:

```
> A := FiniteAffinePlane< 4 | Setseq(Subsets({1, 2, 3, 4}, 2)) >;
> A: Maximal;
Affine Plane of order 2
Points: {@ 1, 2, 3, 4 @}
Lines:
  {1, 3},
  {1, 4},
  {2, 4},
  {2, 3},
  {1, 2},
  {3, 4}
```

To demonstrate the use of the **Check** argument, we recreate the classical projective plane of order 16 with **Check := true** (the default) and **Check := false**.

```
> P, V, L := FiniteProjectivePlane(16);
> time P2 := FiniteProjectivePlane<
>   Points(P) | {Set(1): 1 in L} : Check := true >;
Time: 10.769
> time P2 := FiniteProjectivePlane<
>   Points(P) | {Set(1): 1 in L} : Check := false >;
Time: 0.030
```

---

## 141.3 The Point-Set and Line-Set of a Plane

### 141.3.1 Introduction

An affine or projective plane in MAGMA consists of three objects: the plane  $P$  itself, the *point-set*  $V$  of  $P$ , and the *line-set*  $L$  of  $P$ .

Although called the point-set and line-set,  $V$  and  $L$  are not actual MAGMA sets. They simply act as the parent structures for the points and lines (respectively) of the plane  $P$ , enabling easy creation of these objects via the `!` and `.` operators.

The point-set  $V$  belongs to the MAGMA category `PlanePtSet`, and the line-set  $L$  to the category `PlaneLnSet`.

In this section, the functions used to create point-sets, line-sets and the points and lines themselves are described.

### 141.3.2 Creating Point-Sets and Line-Sets

As mentioned above, the point-set and line-set are returned as the second and third arguments of any function which creates a plane. They can also be created via the following two functions.

`PointSet(P)`

Given a plane  $P$ , return the point-set  $V$  of  $P$ .

`LineSet(P)`

Given a plane  $P$ , return the line-set  $L$  of  $P$ .

### 141.3.3 Using the Point-Set and Line-Set to Create Points and Lines

For efficiency and clarity, the points and lines of a plane are given special types in MAGMA. The category names for points and lines are `PlanePt` and `PlaneLn` respectively. They can be created in the following ways.

`V . i`

Given the point-set  $V$  of a plane  $P$  and an integer  $i$ , return the  $i$ -th point of  $P$ .

`V ! [a, b, c]`

Given the point-set  $V$  of a classical projective plane  $P = PG_2(K)$ , and elements  $a, b, c$  of the finite field  $K$ , create the projective point  $(a : b : c)$  in the plane  $P$ .

`V ! [a, b]`

Given the point-set  $V$  of a classical affine plane  $P = AG_2(K)$ , and elements  $a, b$  of the finite field  $K$ , create the point  $(a, b)$  in the plane  $P$ .

$V ! x$
---------

Given the point-set  $V$  of a plane  $P$ , return the point of  $P$  corresponding to the element  $x$ , which should be coercible into the underlying point set for  $P$ . (In the case of classical planes,  $x$  should be coercible to a vector.)

<code>Representative(V)</code>
--------------------------------

<code>Rep(V)</code>
---------------------

Given the point-set  $V$  of a plane  $P$ , return a representative point of  $P$ .

<code>Random(V)</code>
------------------------

Given the point-set  $V$  of a plane  $P$ , return a random point of  $P$ .

$L . i$
---------

Given the line-set  $L$  of a plane  $P$  and an integer  $i$ , return the  $i$ -th line of  $P$ .

$L ! [a, b, c]$
-----------------

Given the line set  $L$  of a classical plane  $P$  defined over a finite field  $K$ , and elements  $a, b, c$  of  $K$ , create the line  $\langle a : b : c \rangle$  (i.e. the line given by the equation  $ax+by+cz = 0$  if  $P$  is projective, or  $ax + by + c = 0$  if  $P$  is affine).

$L ! [m, b]$
--------------

Given the line set  $L$  of a classical affine plane  $P = AG_2(K)$ , and elements  $m, b$  of the finite field  $K$ , create the affine line  $y = mx + b$  in  $P$ .

$L ! S$
---------

Given the line-set  $L$  of a plane  $P$  and a set or sequence  $S$  of collinear points of  $P$ , return the line containing the points of  $S$ .

$L ! l$
---------

Given the line-set  $L$  of a plane  $P$  and a line  $l$  of a (possibly) different plane (generally a subplane of  $P$ ), return the line of  $P$  corresponding to  $l$ .

<code>Representative(L)</code>
--------------------------------

<code>Rep(L)</code>
---------------------

Given the line-set  $L$  of a plane  $P$ , return a representative line of  $P$ .

<code>Random(L)</code>
------------------------

Given the line-set  $L$  of a plane  $P$ , return a random line of  $P$ .

**Example H141E2**

---

The following example shows how points and lines of a plane can be created. First we study a classical projective plane.

```
> P, V, L := FiniteProjectivePlane(5);
> V;
Point-set of Projective Plane PG(2, 5)
> L;
Line-set of Projective Plane PG(2, 5)
```

Create the third point of  $P$ :

```
> V.3;
( 0 : 0 : 1 )
```

Create the point  $(1 : 2 : 3)$  of  $P$ :

```
> V![1, 2, 3];
( 1 : 2 : 3 )
```

Choose a random point of  $P$ :

```
> Random(V);
( 1 : 0 : 0 )
> Random(V);
( 0 : 0 : 1 )
```

Create the sixth line of  $P$ :

```
> L.6;
< 1 : 1 : 3 >
```

Create the line of  $P$  given by the equation  $4x + 3y + 2z = 0$ :

```
> L![4, 3, 2];
< 1 : 2 : 3 >
```

Create the line of  $P$  containing the points  $(0 : 0 : 1)$  and  $(0 : 1 : 0)$ :

```
> L![ V | [0, 0, 1], [0, 1, 0] ];
< 1 : 0 : 0 >
```

Get a representative from the line-set of  $P$ , and a random line:

```
> Rep(L);
< 1 : 0 : 0 >
> Random(L);
< 1 : 2 : 4 >
```

Now we look at a non-classical plane.

```
> V := {2, 4, 6, 8};
> A, P, L := FiniteAffinePlane< SetToIndexedSet(V) | Setseq(Subsets(V, 2)) >;
> A: Maximal;
Affine Plane of order 2
```

Points: {0 2, 4, 6, 8 0}

Lines:

{6, 8},

{2, 6},

{2, 8},

{2, 4},

{4, 6},

{4, 8}

> P;

Point-set of Affine Plane of order 2

> L;

Line-set of Affine Plane of order 2

Get the third point of  $A$ :

> P.3;

6

Create the point of  $A$  given by the integer 4:

> P!4;

4

Get a representative from the point-set of  $A$ :

> Rep(P);

2

Get the third line of  $A$ :

> L.3;

{2, 8}

Create the line of  $A$  containing the integers 2 and 6:

> L![2, 6];

{2, 6}

Choose a random line from  $A$ :

> Random(L);

{6, 8}

---

### 141.3.4 Retrieving the Plane from Points, Lines, Point-Sets and Line-Sets

The `ParentPlane` function allows you to access the plane to which a point, line, point-set or line-set belongs.

`ParentPlane(V)`

The plane  $P$  for which  $V$  is the point-set.

`ParentPlane(L)`

The plane  $P$  for which  $L$  is the line-set.

`ParentPlane(p)`

The plane  $P$  for which  $p$  is a point.

`ParentPlane(l)`

The plane  $P$  for which  $l$  is a line.

## 141.4 The Set of Points and Set of Lines

It may sometimes be desirable to have an explicitly enumerated set of the points or lines of a plane, as opposed to the point-set and line-set, which aren't true MAGMA sets. The following functions have been provided for this purpose.

`Points(P)`

An indexed set  $E$  whose elements are the points of the plane  $P$ . Note that this creates a standard indexed set and not the point-set of  $P$ , in contrast to the function `PointSet`.

`Lines(P)`

An indexed set containing the lines of the plane  $P$ . In contrast to the function `LineSet`, this function returns the collection of lines of  $P$  in the form of a standard indexed set.

### 141.5 The Defining Points of a Plane

Points of a plane have their own special type in MAGMA (`PlanePt`). However it may sometimes be necessary to return to the objects used to define the points. These are the elements of the indexed set originally used to create the plane, or, in the case of a classical plane, the vector space elements which define the points of the plane.

The functions listed in this section provide the means to do this.

`Support(P)`

An indexed set  $E$  which is the underlying point set of the plane  $P$  (i.e. the elements of the set have their “real” types; they are no longer a “PlanePt” of  $P$ ).

`Support(l)`

The set of underlying points contained in the line  $l$  of a plane  $P$  (i.e. the elements of the set have their “real” types; they are no longer a “PlanePt” of  $P$ ).

`Support(P, p)`

`Support(p)`

The MAGMA object corresponding to the point  $p$  of a plane  $P$ .

---

#### Example H141E3

The following code shows the difference between the `Points` and `Support` functions.

```
> A := FiniteAffinePlane< {@ 3, 4, 5, 6 @} | {3, 4}, {3, 5}, {3, 6},
>                                     {4, 5}, {4, 6}, {5, 6} >;
> Pts := Points(A);
> Supp := Support(A);
> Pts, Supp;
{@ 3, 4, 5, 6 @}
{@ 3, 4, 5, 6 @}

```

These sets look the same, but the elements have different types:

```
> Universe(Pts);
Point-set of Affine Plane of order 2
> Universe(Supp);
Integer Ring

```

The classical plane case is slightly different. Here the support is a set of vectors.

```
> P, V, L := FiniteProjectivePlane(2);
> Points(P);
{@ ( 1 : 0 : 0 ), ( 0 : 1 : 0 ), ( 0 : 0 : 1 ), ( 1 : 1 : 0 ),
  ( 0 : 1 : 1 ), ( 1 : 1 : 1 ), ( 1 : 0 : 1 ) @}
> Support(P);
{@
  (1 0 0),
  (0 1 0),

```

```

      (0 0 1),
      (1 1 0),
      (0 1 1),
      (1 1 1),
      (1 0 1)
@}
> Universe(Points(P));
Point-set of Projective Plane PG(2, 2)
> Universe(Support(P));
Full Vector space of degree 3 over GF(2)
> l := Random(L);
> l;
< 0 : 0 : 1 >
> Support(l);
{
  (1 0 0),
  (0 1 0),
  (1 1 0)
}

```

---

## 141.6 Subplanes

The **sub** constructor allows subplanes of a projective or affine plane to be created. For classical planes, the **SubfieldSubplane** function is also provided.

**sub**< *P* | *L* >

Given a plane  $P$ , construct the subplane of  $P$  generated by the points specified by  $L$ , where  $L$  is a list of one or more items of the following types:

- (a) A point of  $P$ ;
- (b) A set or sequence of points of  $P$ ;
- (c) A subplane of  $P$ ;
- (d) A set or sequence of subplanes of  $P$ .

The set  $S$  of points defined by the list  $L$  must include a quadrangle if  $P$  is a projective plane and three non-collinear points if  $P$  is an affine plane. The function returns the smallest subplane of  $P$  containing  $S$ .

**SubfieldSubplane**( $P$ ,  $F$ )

The plane obtained from the classical plane  $P$  by taking only those points of  $P$  which have all coordinates lying in  $F$ , where  $F$  must be a subfield of **Field**( $P$ ).



**Example H141E4**

In the plane  $PG_2(4)$ , the points  $(1 : 0 : 0)$ ,  $(0 : 1 : 0)$ ,  $(0 : 0 : 1)$  and  $(1 : w : 1)$ , where  $w$  is a primitive element of  $\mathbf{F}_4$ , form a quadrangle. We form the subplane of  $PG_2(4)$  generated by this quadrangle.

```
> K<w> := GF(4);
> P, V, L := FiniteProjectivePlane(K);
> S := sub< P | [ V | [1, 0, 0], [0, 1, 0], [0, 0, 1], [1, w, 1] ] >;
> S: Maximal;
Projective Plane of order 2
Points: { @ ( 1 : 0 : 0 ), ( 0 : 1 : 0 ), ( 0 : 0 : 1 ), ( 1 : w : 0 ),
( 1 : 0 : 1 ), ( 1 : w : 1 ), ( 0 : 1 : w^2 ) @ }
Lines:
  {( 0 : 1 : 0 ), ( 0 : 0 : 1 ), ( 0 : 1 : w^2 )},
  {( 1 : 0 : 0 ), ( 0 : 0 : 1 ), ( 1 : 0 : 1 )},
  {( 1 : 0 : 0 ), ( 0 : 1 : 0 ), ( 1 : w : 0 )},
  {( 1 : 0 : 0 ), ( 1 : w : 1 ), ( 0 : 1 : w^2 )},
  {( 0 : 1 : 0 ), ( 1 : 0 : 1 ), ( 1 : w : 1 )},
  {( 0 : 0 : 1 ), ( 1 : w : 0 ), ( 1 : w : 1 )},
  {( 1 : w : 0 ), ( 1 : 0 : 1 ), ( 0 : 1 : w^2 )}
```

We next form the subplane of  $AG_2(4)$  over  $\mathbf{F}_2$ .

```
> A := FiniteAffinePlane(4);
> S := SubfieldSubplane(A, GF(2));
> S: Maximal;
Affine Plane AG(2, 2)
> S subset A;
true
```

**141.7 Structures Associated with a Plane**

There are a number of structures naturally associated with a plane. This section lists some functions for accessing or creating them.

VectorSpace(P)

The vector space underlying the classical plane  $P$ .

Field(P)

The field over which the classical plane  $P$  is defined.

IncidenceMatrix(P)

The incidence matrix of the plane  $P$ .

Dual(P)

The dual of the projective plane  $P$ .

**Example H141E5**

---

The functions in this section are illustrated by the following example.

```
> A := FiniteAffinePlane(4);
> VectorSpace(A);
Full Vector space of degree 2 over GF(2^2)
> Field(A);
Finite field of size 2^2
>
> P := FiniteProjectivePlane< 7 | {1, 3, 5}, {1, 2, 7}, {1, 4, 6}, {2, 3, 6},
>                                {2, 4, 5}, {3, 4, 7}, {5, 6, 7} >;
> IP := IncidenceMatrix(P);
> IP;
[1 1 1 0 0 0 0]
[0 1 0 1 1 0 0]
[1 0 0 1 0 1 0]
[0 0 1 0 1 1 0]
[1 0 0 0 1 0 1]
[0 0 1 1 0 0 1]
[0 1 0 0 0 1 1]
> D := Dual(P);
> D;
Projective Plane of order 2
> IncidenceMatrix(D) eq Transpose(IP);
true
```

---

**141.8 Numerical Invariants of a Plane**

Order(P)

The order of the plane  $P$ .

NumberOfPoints(P)

#V

The cardinality  $v$  of the point-set  $V$  of the plane  $P$ .

NumberOfLines(P)

#L

The cardinality  $v$  of the line-set  $L$  of the plane  $P$ .

pRank(P)

The  $p$ -rank of the plane  $P$  of order  $p^t$ .

pRank(P, p)

The  $p$ -rank of the plane  $P$ .

**Example H141E6**

---

We demonstrate some of the above functions with the plane  $PG_2(8)$ .

```
> P := FiniteProjectivePlane(8);
> NumberOfLines(P);
73
> Order(P);
8
> pRank(P);
28
> pRank(P, 2);
28
> pRank(P, 3);
72
> pRank(P, 5);
73
```

---

**141.9 Properties of Planes**

IsDesarguesian(P)

Returns **true** if and only if the plane  $P$  is a desarguesian plane.

IsSelfDual(P)

Returns **true** if and only if the projective plane  $P$  is self-dual.

**141.10 Identity and Isomorphism**

Two planes are considered equal if their point sets are equal and they have the same lines.

P eq Q

Return **true** if the planes  $P$  and  $Q$  are equal, otherwise **false**.

P ne Q

Return **true** if the planes  $P$  and  $Q$  are not equal, otherwise **false**.

IsIsomorphic(P, Q: *parameters*)

**AutomorphismGroups**

MONSTGELT

*Default* : “None”

Return **true** if the planes  $P$  and  $Q$  are isomorphic, otherwise **false**. If the planes are isomorphic, an isomorphism  $f$  from  $P$  onto  $Q$  is also returned. The function first computes none, one, or both of the automorphism groups of the left and right planes. This may assist the isomorphism testing. The parameter **AutomorphismGroups**, with valid string values “Both”, “Left”, “Right”, “None”, may be used to specify which of the automorphism groups should be constructed first if not already known. The default is “None”.

<code>P subset Q</code>
-------------------------

Returns `true` if  $P$  is a subplane of  $Q$ , otherwise `false`.

## 141.11 The Connection between Projective and Affine Planes

There exist natural mathematical constructions to form a projective plane from an affine plane and vice versa. The functions in this section provide a quick and easy way to do this in MAGMA.

<code>FiniteAffinePlane(P, l)</code>
--------------------------------------

The affine plane obtained by removing the line  $l$  from the projective plane  $P$ , together with the point set and line set of the affine plane, plus the embedding map from the affine plane to  $P$ .

<code>ProjectiveEmbedding(P)</code>
-------------------------------------

The projective completion of the affine plane  $P$ , together with the point set and line set of the projective plane, plus the embedding map from  $P$  to the projective plane.

### Example H141E7

---

We begin with the classical affine plane  $A$  of order 3, and take the projective embedding  $P$  of  $A$ . We then remove a randomly selected line from  $P$ , and show that the affine plane produced by this action is isomorphic to the original affine plane  $A$ .

```
> A := FiniteAffinePlane(3);
> P := ProjectiveEmbedding(A);
> P;
Projective Plane of order 3
> A2 := FiniteAffinePlane(P, Random(LineSet(P)));
> A2;
Affine Plane of order 3
> iso, map := IsIsomorphic(A, A2);
> is_iso, map := IsIsomorphic(A, A2);
> is_iso;
true
> map;
Mapping from: PlaneAff: A to PlaneAff: A2
```

We demonstrate the use of the embedding map to get the correspondence between the points of the affine and projective planes.

```
> K<w> := GF(4);
> A, AP, AL := FiniteAffinePlane(K);
> P, PP, PL, f := ProjectiveEmbedding(A);
```

Now take a point of the affine plane and map it into the projective.

```
> AP.5;
```

```
( 1, w )
> AP.5 @ f;
5
```

Our point corresponds to `PP.5`, which in the affine plane is the pair  $(1, w)$ . The map `f` can be applied to any point or line of the affine plane to get the corresponding point or line of the projective plane. Given any point or line of the projective plane, provided that it is not on the adjoined line at infinity, the preimage in the affine plane can be found.

The line at infinity is always the last line in the line set of the projective plane created by `ProjectiveEmbedding`. We will call this line `linf`:

```
> linf := PL.#PL;
> linf;
{17, 18, 19, 20, 21}
> SetSeed(1, 3);
> p := Random(PP);
> p in linf;
false
> p @@ f;
( w, 1 )
> l := Random(PL);
> l eq linf;
false
> l @@ f;
< 1 : 1 : 0 >
> $1 @ f eq l;
true
```

Since neither `p` nor `l` were infinite we could find their preimages under `f`. Of course, when we map a line from `P` to `A` and back, we get the line we started with.

When an embedding is constructed by `FiniteAffinePlane(P, l)`, then `l` is the line at infinity for this embedding.

## 141.12 Operations on Points and Lines

All the usual equality, membership and subset functions are provided along with a collection of deconstruction functions and others.

### 141.12.1 Elementary Operations

`p eq q`

Returns `true` if the points  $p$  and  $q$  are equal, otherwise `false`.

`p ne q`

Return `true` if the points  $p$  and  $q$  are not equal, otherwise `false`.

$l \text{ eq } m$

Return **true** if the lines  $l$  and  $m$  are equal, otherwise **false**.

$l \text{ ne } m$

Return **true** if the lines  $l$  and  $m$  are not equal, otherwise **false**.

$p \text{ in } l$

Return **true** if point  $p$  lies on the line  $l$ , otherwise **false**.

$p \text{ notin } l$

Return **true** if point  $p$  does not lie on the line  $l$ , otherwise **false**.

$S \text{ subset } l$

Given a subset  $S$  of the point set of the plane  $P$  and a line  $l$  of  $P$ , return **true** if the subset  $S$  of points lies on the line  $l$ , otherwise **false**.

$S \text{ notsubset } l$

Given a subset  $S$  of the point set of the plane  $P$  and a line  $l$  of  $P$ , return **true** if the subset  $S$  of points does not lie on the line  $l$ , otherwise **false**.

$l \text{ meet } m$

The unique point common to the lines  $l$  and  $m$ .

$\text{Representative}(l)$

$\text{Rep}(l)$

Given a line  $l$  of the plane  $P$ , return a representative point of  $P$  which is incident with  $l$ .

$\text{Random}(l)$

Given a line  $l$  of the plane  $P$ , return a random point of  $P$  which is incident with  $l$ .

## 141.12.2 Deconstruction Functions

$\text{Index}(P, p)$

Given a point  $p$  from the point-set  $V$  of a plane  $P$ , return the index of  $p$ , i.e. the integer  $i$  such that  $p$  is  $V.i$ .

$\text{Index}(P, l)$

Given a line  $l$ , return the index of  $l$  in the plane  $P$ , i.e. the integer  $i$  such that  $l$  is  $L.i$  (where  $L$  is the line-set of  $P$ ).

$p[i]$

The  $i$ -th coordinate of the point  $p$ , which must be from a classical plane. If  $p$  is from a projective plane, then  $i$  must satisfy  $1 \leq i \leq 3$ ; if  $p$  is from an affine plane, then  $i$  must satisfy  $1 \leq i \leq 2$ .

l[i]

The  $i$ -th coordinate of the line  $l$ , which must be from a classical plane. The integer  $i$  must satisfy  $1 \leq i \leq 3$ . Recall that in a classical plane  $\langle a : b : c \rangle$  (where  $a, b, c \in K$ ) represents the line given by the equation  $ax + by + cz = 0$  in a projective plane or  $ax + by + c = 0$  in an affine plane.

Coordinates(P, p)

Given a point  $p = (a : b : c)$  from a classical projective plane  $P$  (or  $p = (a, b)$  from a classical affine plane  $P$ ), return the sequence  $[a, b, c]$  (or  $[a, b]$  in the affine case) of coordinates of  $p$ .

Coordinates(P, l)

Given a line  $l = \langle a : b : c \rangle$  from a classical plane  $P$  (projective or affine), return the sequence  $[a, b, c]$  of coordinates of  $l$ .

ElementToSequence(p)

Eltseq(p)

Given a point  $p = (a : b : c)$  from a classical projective plane  $P$  (or  $p = (a, b)$  from a classical affine plane  $P$ ), return the sequence  $[a, b, c]$  (or  $[a, b]$  in the affine case) of coordinates of  $p$ .

ElementToSequence(l)

Eltseq(l)

Given a line  $l = \langle a : b : c \rangle$  from a classical plane  $P$  (projective or affine), return the sequence  $[a, b, c]$  of coordinates of  $l$ .

Set(l)

The set of points contained in the line  $l$ .

---

### Example H141E8

The following example illustrates the use of some of the elementary and deconstruction functions on lines and points discussed in the previous two subsections.

```
> K<w> := GF(4);
> P, V, L := FiniteProjectivePlane(K);
```

Create the line  $x + z = 0$ :

```
> l := L![1, 0, 1];
> l;
< 1 : 0 : 1 >
```

Look at the points on the line  $l$ :

```
> Set(l);
{ ( 0 : 1 : 0 ), ( 1 : w^2 : 1 ), ( 1 : 0 : 1 ),
```

$$(1 : w : 1), (1 : 1 : 1) \}$$

Get the coordinates of the line  $l$ :

```
> Coordinates(P, l);
[ 1, 0, 1 ]
> l[1];
1
```

Find the index of the line  $l$  in the line-set  $L$  of  $P$ , and check it:

```
> Index(P, l);
8
> l eq L.8;
true
```

Test if a point is on the line  $l$ :

```
> V![1, 0, 1] in l;
true
```

Test a set of points for containment in  $l$ :

```
> S := {V.1, V.2};
> S;
{ (1 : 0 : 0), (0 : 1 : 0) }
> S subset l;
false
```

Create the line containing the points in  $S$ :

```
> l2 := L!S;
> l2;
< 0 : 0 : 1 >
> S subset l2;
true
```

And finally, find the point common to the lines  $l$  and  $l2$ :

```
> p := l meet l2;
> p;
( 0 : 1 : 0 )
> p[3];
0
```

---



### 141.12.3 Other Point and Line Functions

**IsCollinear**( $P$ ,  $S$ )

Return **true** if the set  $S$  of points of the plane  $P$  are collinear, otherwise **false**. If the points are collinear, the line which they define is also returned.

**IsConcurrent**( $P$ ,  $R$ )

Return **true** if the set  $R$  of lines of the plane  $P$  are concurrent, otherwise **false**. If the lines are concurrent, their common point is returned as a second value.

**ContainsQuadrangle**( $P$ ,  $S$ )

Return **true** if the set  $S$  of points of a plane  $P$  contains a quadrangle.

**Pencil**( $P$ ,  $p$ )

The pencil of lines passing through the point  $p$  in the plane  $P$ .

**Slope**( $l$ )

The slope of the line  $l$  of a classical affine plane  $P$ .

**IsParallel**( $P$ ,  $l$ ,  $m$ )

Return **true** if the line  $l$  is parallel to the line  $m$  in the affine plane  $P$ .

**ParallelClass**( $P$ ,  $l$ )

The parallel class containing the line  $l$  of an affine plane  $P$ .

**ParallelClasses**( $P$ )

The partition into parallel classes of the lines of the affine plane  $P$ .

#### Example H141E9

---

We use the affine plane  $AG_2(3)$  to demonstrate some of the above functions.

```
> A, V, L := FiniteAffinePlane(3);
```

Create the line  $y = 2x + 1$  in  $A$ , and check its slope:

```
> l := L![2, 1];
> l;
< 1 : 1 : 2 >
> Slope(l);
2
```

Find the lines parallel to  $l$ :

```
> ParallelClass(l);
{
  < 1 : 1 : 0 >,
  < 1 : 1 : 1 >,
  < 1 : 1 : 2 >
```

```

}
> [Slope(m): m in ParallelClass(l)];
[ 2, 2, 2 ]

```

Get the pencil of lines through a point of  $l$ :

```

> p := Rep(l);
> p;
( 1, 0 )
> Pencil(A, p);
{
  < 1 : 0 : 2 >,
  < 1 : 1 : 2 >,
  < 1 : 2 : 2 >,
  < 0 : 1 : 0 >
}

```

---

### 141.13 Arcs

A  $k$ -arc in a projective or affine plane  $P$  is a set of  $k$  points of  $P$ , no three of which are collinear. A  $k$ -arc is *complete* if it cannot be extended to a  $(k+1)$ -arc by the addition of another point. A *tangent* to an arc  $A$  is a line which meets  $A$  exactly once; a *secant* is a line which meets  $A$  exactly twice; and a *passant*, or *external line*, is a line which does not meet  $A$  at all.

kArc(P, k)

Return a  $k$ -arc for the plane  $P$ .

CompleteKArc(P, k)

Return a complete  $k$ -arc for the plane  $P$  (if one exists).

IsArc(P, A)

Returns **true** if the set of points  $A$  is an arc in the plane  $P$ , i.e. no three points of  $A$  are collinear.

IsComplete(P, A)

Returns **true** if the  $k$ -arc  $A$  is complete in the plane  $P$ .

Conic(P, S)

Given a set  $S$  of five points belonging to a classical projective plane  $P$  of order  $n > 3$  and being in general position, construct the unique conic that passes through them.

**QuadraticForm(S)**

Given a set  $S$  of five points belonging to a classical projective plane of order  $n > 3$  that are in general position, return the quadratic form defining the conic containing the five points.

**Tangent(P, A, p)**

Given an arc  $A$  in the plane  $P$ , and a point  $p$  on  $A$ , return a tangent to  $A$  at  $p$ .

**AllTangents(P, A)**

Given an arc  $A$  in the plane  $P$ , return the set of tangent lines to  $A$ .

**AllSecants(P, A)**

Given an arc  $A$  in the plane  $P$ , return the set of secant lines to  $A$ .

**ExternalLines(P, A)****AllPassants(P, A)**

Given an arc  $A$  in the plane  $P$ , return the set of external lines to  $A$ .

**Knot(P, C)**

Given a conic  $C$  in the projective plane  $P$  of even order, return the knot of the conic  $C$ , i.e. the intersection point of the tangents to  $C$ .

**Exterior(P, C)**

Given a conic  $C$  in the projective plane  $P$  of odd order, return the exterior points of  $C$ , i.e. the points of  $P$  that lie on two tangents of  $C$ .

**Interior(P, C)**

Given a conic  $C$  in the projective plane  $P$  of odd order, return the interior points of  $C$ , i.e. the points of  $P$  that do not lie on any tangent of  $C$ .

**Example H141E10**

---

The following sequence of instructions constructs an oval design from  $PG_2(16)$ .

```
> P, V, L := FiniteProjectivePlane(16);
> oval := kArc(P, 18);
> pts := Points(P) diff oval;
> lns := ExternalLines(P, oval);
> I := IncidenceStructure< SetToIndexedSet(pts) | [l meet pts : l in lns] >;
> D := Design(Dual(I), 2);
> D;
2-(120, 8, 1) Design with 255 blocks
```

The next example uses various functions discussed so far, and shows the relationship between a plane and its subplanes.

```
> K<w> := GF(9);
```

```

> P, V, L := FiniteProjectivePlane(K);
> c := kArc(P, 5);
> c;
{ ( 1 : 0 : 0 ), ( 0 : 1 : 0 ), ( 0 : 0 : 1 ), ( 1 : 2 : w^3 ),
  ( 1 : w : w ) }
> C := Conic(P, c);
> C;
{ ( 1 : 0 : 0 ), ( 0 : 1 : 0 ), ( 0 : 0 : 1 ), ( 1 : 2 : w^3 ),
  ( 1 : w : w ), ( 1 : w^5 : w^6 ), ( 1 : w^7 : 2 ), ( 1 : w^2 : 1 ),
  ( 1 : w^3 : w^5 ), ( 1 : w^6 : w^2 ) }
> #C;
10
> #Interior(P, C);
36
>
> S, SV, SL := SubfieldSubplane(P, GF(3));
> S subset P;
true
> a := kArc(S, 4);
> IsArc(S, a);
true
> IsArc(P, a);
true
> IsComplete(S, a);
true
> IsComplete(P, a);
false
> a;
{ ( 1 : 0 : 0 ), ( 0 : 1 : 0 ), ( 0 : 0 : 1 ), ( 1 : 2 : 1 ) }
> S2 := sub< P | a >;
> S2;
Projective Plane of order 3
> S2 eq S;
true
> p := Random(a);
> p;
( 1 : 2 : 1 )
> Tangent(S, a, p);
< 1 : 2 : 1 >
> AllTangents(S, a);
{
  < 1 : 2 : 0 >,
  < 0 : 1 : 2 >,
  < 1 : 2 : 1 >,
  < 1 : 0 : 1 >
}
> AllTangents(P, a);
{

```

```

    < 1 : w^5 : w^6 >,
    < 1 : 0 : w >,
    < 1 : w^2 : w >,
    < 1 : 0 : w^6 >,
    < 1 : 0 : 1 >,
    < 1 : w^6 : w^3 >,
    < 1 : w^6 : 0 >,
    < 0 : 1 : w^6 >,
    < 1 : 2 : 0 >,
    < 0 : 1 : 2 >,
    < 1 : 2 : 1 >,
    < 1 : w^3 : w^5 >,
    < 1 : 0 : w^7 >,
    < 1 : 0 : w^2 >,
    < 1 : w : 0 >,
    < 0 : 1 : w >,
    < 1 : 0 : w^5 >,
    < 1 : 0 : w^3 >,
    < 1 : w : w^7 >,
    < 1 : w^3 : 0 >,
    < 0 : 1 : w^3 >,
    < 1 : w^7 : w^2 >,
    < 1 : w^5 : 0 >,
    < 0 : 1 : w^5 >,
    < 1 : w^7 : 0 >,
    < 0 : 1 : w^7 >,
    < 1 : w^2 : 0 >,
    < 0 : 1 : w^2 >
}

```

### 141.14 Unitals

A unital in the classical projective plane  $PG_2(q^2)$  is a set of  $q^3 + 1$  points such that every line meeting two of these points meets exactly  $q + 1$  of them.

**IsUnital**( $P$ ,  $U$ )

Given a set of points  $U$  belonging to a projective plane  $P$  defined over a field of cardinality  $q^2$ , return **true** if  $U$  is a unital.

**AllTangents**( $P$ ,  $U$ )

Given a unital set of points  $U$  in the projective plane  $P$ , return the set of tangents to the points of  $U$ .

**UnitalFeet**( $P$ ,  $U$ ,  $p$ )

The set of intersections of the unital set of points  $U$  with the tangents to  $U$  in the plane  $P$  which pass through the point  $p$ .

**Example H141E11**

The following code computes the Hermitian unital given by the equation  $x^{q+1} + y^{q+1} + z^{q+1} = 0$  in  $PG_2(q^2)$  for  $q = 3$ .

```
> q := 3;
> F<w> := GaloisField(q ^ 2);
> P, V, L := FiniteProjectivePlane(F);
>
> hu := { V | [x,y,z] : x, y, z in F |
>           x^(q+1) + y^(q+1) + z^(q+1) eq 0 and {x, y, z} ne {0} };
>
> IsUnital(P, hu);
true
> UnitalFeet(P, hu, V.1);
{ ( 0 : 1 : w ), ( 0 : 1 : w^3 ), ( 0 : 1 : w^5 ), ( 0 : 1 : w^7 ) }
```

Since this set has more than one element,  $V.1$  must not be in  $hu$ :

```
> V.1 in hu;
false
```

For a point in  $hu$ :

```
> UnitalFeet(P, hu, Rep(hu));
{ ( 1 : 0 : w^7 ) }
```

Now we construct the design given by  $hu$ .

```
> blks := [blk : lin in L | #blk eq (q+1) where blk is lin meet hu ];
> D := Design< 2, SetToIndexedSet(hu) | blks >;
> D;
2-(28, 4, 1) Design with 63 blocks
```

## 141.15 The Collineation Group of a Plane

The automorphism group (or collineation group)  $A$  of a plane  $P$  is always presented as a permutation group  $G$  acting on the standard support  $\{1, ..v\}$ , where  $v$  is the number of points of  $P$ . The reasons for this include the fact that if the group is represented as acting on a set of objects relating to the plane, printed permutations are often unreadable.

So the collineation group  $G$  of  $P$  does not act directly on  $P$ . Instead,  $G$ -sets are used to transfer the action of  $G$  to various sets associated with  $P$ . The two most important  $G$ -sets, corresponding to action of  $G$  on the point set and on the line set, are returned by each of the functions provided for constructing the collineation group or some specified subgroup of it.

In some circumstances, rather than viewing collineations as group elements, it is desirable to view them as mappings of  $P$  into itself. Associated with each incidence structure is a mapping structure,  $\text{Aut}(P)$ , which denotes the set of collineations of  $P$ . Note that  $\text{Aut}(P)$  is the *parent* of the collineations of  $G$  so that the function  $\text{Aut}(P)$  simply creates a

shell structure rather than the actual collineation group of  $P$ . A transfer map is provided to convert a permutation of the collineation group  $G$  into a mapping belonging to  $\text{Aut}(P)$ .

### 141.15.1 The Collineation Group Function

<code>CollineationGroup(P)</code>
-----------------------------------

<code>AutomorphismGroup(P)</code>
-----------------------------------

<code>PointGroup(P)</code>
----------------------------

Construct the collineation group  $G$  of the plane  $P$ . The group  $G$  is returned as a permutation group on the standard support  $\{1 \dots v\}$ , where  $v$  is the number of points of  $P$ . The function also returns: a  $G$ -set  $Y$  being the point set of  $P$  acted on by  $G$ ; a  $G$ -set  $W$  being the line set of  $P$  acted on by  $G$ ; a power structure  $S$ ; a transfer map  $t$ . Given a permutation  $g$  from  $G$ , one can create a map  $f = t(g)$  which represents the automorphism  $g$  as a mapping in  $S$  from  $P$  to  $P$  (which maps both point sets and line sets). The  $G$ -sets  $Y$  and  $W$  should be used if one wishes to compute stabilizers or similar such subgroups of  $G$  so that the appropriate action is used.

<code>LineGroup(P)</code>
---------------------------

Construct the collineation group  $G$  of the plane  $P$  in its action on the lines of  $P$ . The group  $G$  is returned as a permutation group on the standard support  $\{1 \dots l\}$ , where  $l$  is the number of lines of  $P$ . A power structure  $S$  and transfer map  $t$  are also returned, so that, given a permutation  $g$  from  $G$ , one can create a map  $f = t(g)$  which represents the automorphism  $g$  as a mapping in  $S$  from  $L$  to  $L$ , where  $L$  is the line set of  $P$ .

<code>CollineationGroupStabilizer(P, k)</code>
--

A subgroup  $G$  of the collineation group of the plane  $P$  which stabilizes the first  $k$  base points, together with the points  $G$ -set, the lines  $G$ -set, the power structure  $A$  of all automorphisms of  $P$ , and the transfer map  $t$  from  $G$  into  $A$ .

<code>CollineationSubgroup(P)</code>
--------------------------------------

A subgroup  $G$  of the collineation group of the plane  $P$  generated by one element, together with the points  $G$ -set, the lines  $G$ -set, the power structure  $A$  of all automorphisms of  $P$ , and the transfer map  $t$  from  $G$  into  $A$ .

### 141.15.2 General Action of Collineations

The collineation group  $G$  of a plane  $P$  is given in its action on the standard support. This support may be regarded as the indices of the points of  $P$ . The action of  $G$  on  $P$  is obtained using the  $G$ -set mechanism. The two basic  $G$ -sets associated with  $P$  correspond to the action of  $G$  on the set of points  $V$  and the set of lines  $L$  of  $P$ . These two  $G$ -sets are given as return values of the function `AutomorphismGroup`. Additional  $G$ -sets associated with  $P$  may be built using the  $G$ -set constructors. Given a  $G$ -set  $Y$  for  $G$ , the action of  $G$  on  $Y$  may be studied using the permutation group functions that allow a  $G$ -set as an argument. In this section, only a few of the available functions are described: see the chapter on permutation groups for a complete list.

The action of the collineation group on the plane may also be obtained using the  $\hat{\phantom{x}}$  operator.

$y \hat{g}$
-------------

Let  $G$  be a subgroup of the collineation group for the plane  $P$  and suppose  $g \in G$ . Given an element  $y$  that is either a point or line of  $P$ , return the image of  $y$  under  $g$ .

$y \hat{G}$
-------------

Let  $G$  be a subgroup of the collineation group for the plane  $P$ . Given an element  $y$  that is either a point or line of  $P$ , return the orbit of  $y$  under  $G$ .

<code>Image(g, Y, y)</code>
-----------------------------

Let  $G$  be a subgroup of the collineation group for the plane  $P$  and let  $Y$  be a  $G$ -set for  $G$ . Given an element  $y$  belonging either to  $Y$  or to a  $G$ -set derived from  $Y$ , find the image of  $y$  under  $G$ .

<code>Orbit(G, Y, y)</code>
-----------------------------

Let  $G$  be a subgroup of the collineation group for the plane  $P$  and let  $Y$  be a  $G$ -set for  $G$ . Given an element  $y$  belonging either to  $Y$  or to a  $G$ -set derived from  $Y$ , construct the orbit of  $y$  under  $G$ .

<code>Orbits(G, Y)</code>
---------------------------

Let  $G$  be a subgroup of the collineation group for the plane  $P$  and let  $Y$  be a  $G$ -set for  $G$ . This function constructs the orbits of the action of  $G$  on  $Y$ .

<code>Stabilizer(G, Y, y)</code>
----------------------------------

Let  $G$  be a subgroup of the collineation group for the plane  $P$  and let  $Y$  be a  $G$ -set for  $G$ . Given an element  $y$  belonging either to  $Y$  or to a  $G$ -set derived from  $Y$ , construct the stabilizer of  $y$  in  $G$ .



**Action( $G$ ,  $Y$ )**

Given a subgroup  $G$  of the collineation group of the plane  $P$ , and a  $G$ -set  $Y$  for  $G$ , construct the homomorphism  $\phi : G \rightarrow L$ , where the permutation group  $L$  gives the action of  $G$  on the set  $Y$ . The function returns:

- (a) The natural homomorphism  $\phi : G \rightarrow L$ ;
- (b) The induced group  $L$ ;
- (c) The kernel of the action (a subgroup of  $G$ ).

**ActionImage( $G$ ,  $Y$ )**

Given a subgroup  $G$  of the collineation group of the plane  $P$ , and a  $G$ -set  $Y$  for  $G$ , construct the permutation group  $L$  giving the action of  $G$  on the set  $Y$ .

**ActionKernel( $G$ ,  $Y$ )**

Given a subgroup  $G$  of the collineation group of the plane  $P$ , and a  $G$ -set  $Y$  for  $G$ , construct the kernel of the action of  $G$  on the set  $Y$ .

**Example H141E12**

---

We illustrate the use of the  $G$ -sets returned by the `CollineationGroup` function.

```
> P := FiniteProjectivePlane(3);
> G, Y, W := CollineationGroup(P);

Compute the stabilizer of the first point of P:

> H := Stabilizer(G, Y, Points(P)[1]);
> H;
Permutation group H acting on a set of cardinality 13
(3, 9)(5, 7)(6, 11)(8, 12)
(2, 8, 12)(3, 9)(4, 6, 7, 10, 5, 11)
(2, 9)(3, 4)(5, 8)(10, 13)
(2, 8)(3, 9)(4, 5)(6, 10)
(2, 3)(4, 13)(6, 8)(9, 10)
(2, 13, 8)(3, 6, 4)(5, 10, 9)
(2, 3)(4, 9)(7, 12)(10, 13)
> H eq CollineationGroupStabilizer(P, 1);
true
> lines := {m : m in Lines(P) | Points(P)[1] in m};
> l := Random(lines);
> l ^ H;
GSet{
  < 0 : 1 : 0 >,
  < 0 : 0 : 1 >,
  < 0 : 1 : 2 >,
  < 0 : 1 : 1 >
```

```
}

```

Compute the stabilizer of the first line of P:

```
> Stabilizer(G, W, Lines(P)[1]);
Permutation group acting on a set of cardinality 13
(1, 8, 12, 4, 7, 9)(2, 3, 5)(6, 13)
(3, 5, 11)(6, 7, 9)(8, 12, 13)
(4, 10)(5, 11)(6, 7)(8, 12)
(1, 10, 6, 7)(2, 11)(3, 5)(4, 13, 9, 12)
(5, 11)(6, 12)(7, 8)(9, 13)
(1, 10)(3, 5)(6, 7)(12, 13)
(1, 4, 10)(3, 11)(6, 13, 9, 8, 7, 12)
```

### Example H141E13

---

The following function `Bundle` returns a projective bundle in  $PG_2(q)$ .

```
> Bundle := function(q)
>   K<w> := GF(q^3);
>   P, V, L := FiniteProjectivePlane(q^3);
>   G, Y := CollineationGroup(P);
>   S := Support(P); // normalized vectors
>   sig := sub< G |
>       [Index(P, V ! [S[i][1], S[i][2]^q, S[i][3]^q]) : i in [1..#V]]>;
>       // group of planar collineations of order 3
>   p := V ! [1, w^2, w];
>   T := Orbit(sig, Y, p);
>   e2 := V![0, 1, 0];
>   e3 := V![0, 0, 1];
>   S := Points(SubfieldSubplane(P, GF(q)));
>   c23 := Conic(P, T join {e2, e3}) meet S;
>   e1 := Rep(S diff c23);
>   c12 := (Conic(P, T join {e1, e2}) meet S) diff { e1 };
>   c13 := (Conic(P, T join {e1, e3}) meet S) diff { e1 };
>   bundle := [ Conic(P, T join {e1, e}) meet S : e in c23 ] cat
>       [ Conic(P, T join {v1, v2}) meet S : v2 in c13, v1 in c12 ];
>   return FiniteProjectivePlane< S | bundle >;
>
> end function;
>
> PB := Bundle(3);
> PB;
Projective Plane of order 3
```

**Example H141E14**

---

The function `BaerDerivation` below uses a Baer subplane to construct an affine plane.

```
> BaerDerivation := function(q)
> //-----
> // Construct an affine plane by the technique of derivation using
> // Baer subplanes
>
>   Fq2< w > := FiniteField(q^2);
>   V := VectorSpace(Fq2, 3);
>   Plane, Pts, Lns := FiniteProjectivePlane(V);
>   G, Y := CollineationGroup(Plane);
```

Construct a Baer subplane:

```
>   Subplane := SubfieldSubplane(Plane, GF(q));
```

The Baer segment consists of those points of the Baer subplane that lie on the line at infinity. Take the line  $x = 0$  as the line at infinity.

```
>   LineInf := Lns![1, 0, 0];
>
>   BaerSeg := Points(Subplane) meet LineInf;
```

We now find the subgroup of the collineation group that fixes the Baer segment. The translates of the Baer subplane under this subgroup will give us those Baer subplanes that contain the set `BaerSeg`. We use the  $G$ -set `Y` to specify the action of  $G$  on the points of `Plane`.

```
>   StabSeg := Stabilizer(G, Y, BaerSeg);
```

Rather than computing the translates of the entire Baer subplane, we compute the translates of `Subplane - BaerSeg` so that we get exactly those sets which become new affine lines.

```
>   BaerLines := Orbit(StabSeg, Y, Points(Subplane) diff BaerSeg);
```

We complete the new plane by taking those lines of  $PG(2, q^2)$  which intersect the line at infinity at points other than those in the Baer segment. Upon removing the intersection point with `LineInf`, each such line becomes a line of the new affine plane.

```
>   AffLines := BaerLines join { Set(l) diff LineInf : l in Lns |
>                               (BaerSeg meet l) eq {} };
>   return FiniteAffinePlane< SetToIndexedSet(&join(AffLines)) | Setseq(AffLines)
>                               : Check := false >;
> end function; /*BaerDerivation*/
```

---

### 141.15.3 Central Collineations

Let  $p$  be a point and  $l$  a line of a projective plane  $P$ . A  $(p, l)$ -central collineation is a collineation  $\alpha$  of  $P$  which fixes  $l$  pointwise and  $p$  linewise. The line  $l$  is called the axis of  $\alpha$  and the point  $p$  is called the centre of  $\alpha$ .

**CentralCollineationGroup(P, p, l)**

The group  $G$  of  $(p, l)$ -central collineations of a projective plane  $P$ . A power structure  $S$  and transfer map  $t$  are also returned, so that, given a permutation  $g$  from  $G$ , one can create a map  $f = t(g)$  which represents the permutation  $g$  as a mapping in  $S$  from  $P$  to  $P$ . (which maps both point sets and line sets).

**CentralCollineationGroup(P, p)**

The group of central collineations with centre  $p$  of a projective plane  $P$ . A power structure  $S$  and transfer map  $t$  are also returned, so that, given a permutation  $g$  from  $G$ , one can create a map  $f = t(g)$  which represents the permutation  $g$  as a mapping in  $S$  from  $P$  to  $P$ . (which maps both point sets and line sets).

**CentralCollineationGroup(P, l)**

The group of central collineations with axis  $l$  of a projective plane  $P$ . A power structure  $S$  and transfer map  $t$  are also returned, so that, given a permutation  $g$  from  $G$ , one can create a map  $f = t(g)$  which represents the permutation  $g$  as a mapping in  $S$  from  $P$  to  $P$ . (which maps both point sets and line sets).

**IsCentralCollineation(P, g)**

Returns **true** iff the collineation  $g$  of the projective plane  $P$  is a central collineation; if **true**, also returns the centre and axis of  $g$ . The support of the parent of  $g$  must be the point set of  $P$  or the standard support  $\{1 \dots v\}$ , where  $v$  is the number of points of  $P$ .

#### Example H141E15

We find a group of central collineations of a plane  $P$ , and check that a random element of the group is in fact a central collineation and has the correct axis and centre.

```
> P, V, L := FiniteProjectivePlane< 13 |
>   {1, 2, 3, 4}, {1, 5, 6, 7}, {1, 8, 9, 10},
>   {1, 11, 12, 13}, {2, 5, 8, 11}, {2, 6, 9, 12},
>   {2, 7, 10, 13}, {3, 5, 9, 13}, {3, 6, 10, 11},
>   {3, 7, 8, 12}, {4, 5, 10, 12}, {4, 6, 8, 13},
>   {4, 7, 9, 11} >;
> p := V!3;
> l := L.1;
> G := CentralCollineationGroup(P, p, l);
> G;
Permutation group G acting on a set of cardinality 13
Order = 3
```

```

      (5, 13, 9)(6, 11, 10)(7, 12, 8)
> g := Random(G);
> g;
(5, 9, 13)(6, 10, 11)(7, 8, 12)
> is_cent_coll, centre, axis := IsCentralCollineation(P, g);
> is_cent_coll;
true
> centre eq p;
true
> axis eq l;
true

```

Any line through the centre of a central collineation must be fixed by the collineation.

```

> lines := {m : m in Lines(P) | p in m};
> m := Random(lines);
> m;
{3, 7, 8, 12}
> m ^ G;
GSet{
  {3, 7, 8, 12}
}

```

#### 141.15.4 Transitivity Properties

IsPointTransitive(P)

IsTransitive(P)

Return **true** iff the collineation group of the plane  $P$  acts transitively on the points of  $P$ .

IsLineTransitive(P)

Return **true** iff the collineation group of the plane  $P$  acts transitively on the lines of  $P$ .

#### Example H141E16

We check the transitivity of the collineation group of  $AG_2(4)$ .

```

> P := FiniteAffinePlane(4);
> IsPointTransitive(P);
true
> IsLineTransitive(P);
true

```

## 141.16 Translation Planes

The following functions may be used to construct translation planes by derivation. The original code is due to Jenny Key.

**BaerDerivation(q2)**

An affine plane constructed by the technique of derivation with respect to a Baer subplane, where  $q2$  is an even power of a prime.

**BaerSubplane(P)**

A Baer subplane of the projective plane  $P$ .

**OvalDerivation(q: parameters)**

**HallOval**                      **BOOLELT**                      *Default : false*

**Print**                      **BOOLELT**                      *Default : false*

A translation plane from  $\text{PG}(2, q)$ , where  $q$  is a power of 2, computed by derivation with respect to an oval. By default, the oval chosen is that defined by the points of the conic  $y^2 = xz$  together with the nucleus, in  $\text{PG}(2, q)$ . In the case  $q = 16$ , a Hall oval in the plane may be requested by assigning the parameter **HallOval** the value **true**. If the parameter **Print** is assigned the value **true**, then some information is printed during the computation.

## 141.17 Planes and Designs

Projective and affine planes can be viewed as special kinds of designs. The following functions convert between designs and planes.

**Design(P)**

The design corresponding to the points and lines of the plane  $P$ .

**FiniteAffinePlane(D)**

The affine plane corresponding to the incidence structure  $D$ .

**FiniteProjectivePlane(D)**

The projective plane corresponding to the incidence structure  $D$ .

**Example H141E17**

---

The development of a Singer difference set provides a design which satisfies the projective plane axioms, and thus can be converted to a projective plane in MAGMA.

```
> sds := SingerDifferenceSet(2, 3);
> sds;
{ 0, 1, 3, 9 }
> sdv := Development(sds);
> sdv;
2-(13, 4, 1) Design with 13 blocks
> spp := FiniteProjectivePlane(sdv);
> spp: Maximal;
Projective Plane of order 3
Points: {@ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12 @}
Lines:
    {0, 1, 3, 9},
    {1, 2, 4, 10},
    {2, 3, 5, 11},
    {3, 4, 6, 12},
    {0, 4, 5, 7},
    {1, 5, 6, 8},
    {2, 6, 7, 9},
    {3, 7, 8, 10},
    {4, 8, 9, 11},
    {5, 9, 10, 12},
    {0, 6, 10, 11},
    {1, 7, 11, 12},
    {0, 2, 8, 12}
> Universe(Support(spp));
Residue class ring of integers modulo 13
```

---

**141.18 Planes, Graphs and Codes**

MAGMA provides the following functions to create the graphs and codes naturally associated to a projective or affine plane.

LineGraph(P)

The line graph of the plane  $P$ .

IncidenceGraph(P)

The incidence graph of the plane  $P$ . This bipartite graph has as vertex set the union of the point set  $V$  and line set  $L$  of  $P$ . A vertex  $p \in V$  is adjacent to a vertex  $l \in L$  whenever  $p \in l$ .

## LinearCode(P, K)

Given a plane  $P$  with  $v$  points and a finite field  $K$ , this function returns the linear code  $C$  of length  $v$  generated by the characteristic functions of the lines of  $P$  considered as vectors of the  $K$ -space  $K^{(v)}$ .

**Example H141E18**

---

```
> P, V, L := FiniteAffinePlane(3);
> #V, #L;
9 12
> IncidenceGraph(P);
Graph
Vertex  Neighbours
  1      10 11 12 19 ;
  2      16 17 18 19 ;
  3      13 14 15 19 ;
  4      10 15 17 21 ;
  5      12 14 16 21 ;
  6      11 13 18 21 ;
  7      10 14 18 20 ;
  8      11 15 16 20 ;
  9      12 13 17 20 ;
 10      1 4 7 ;
 11      1 6 8 ;
 12      1 5 9 ;
 13      3 6 9 ;
 14      3 5 7 ;
 15      3 4 8 ;
 16      2 5 8 ;
 17      2 4 9 ;
 18      2 6 7 ;
 19      1 2 3 ;
 20      7 8 9 ;
 21      4 5 6 ;
> LinearCode(P, Field(P));
[9, 6, 3] Linear Code over GF(3)
Generator matrix:
[1 0 0 0 0 1 0 1 0]
[0 1 0 0 0 1 0 2 2]
[0 0 1 0 0 1 0 0 1]
[0 0 0 1 0 2 0 1 2]
[0 0 0 0 1 2 0 2 1]
[0 0 0 0 0 0 1 1 1]
```

---



# 142 INCIDENCE GEOMETRY

<b>142.1 Introduction . . . . .</b>	<b>4751</b>	<b>142.8 Shadow Spaces . . . . .</b>	<b>4763</b>
<b>142.2 Construction of Incidence and Coset Geometries . . . . .</b>	<b>4752</b>	ShadowSpace(D, I)	4763
142.2.1 Construction of an Incidence Geometry . . . . .	4752	<b>142.9 Automorphism Group and Correlation Group . . . . .</b>	<b>4764</b>
IncidenceGeometry(G)	4752	AutomorphismGroup(D)	4764
142.2.2 Construction of a Coset Geometry	4756	CorrelationGroup(D)	4764
CosetGeometry(G, S, I)	4756	<b>142.10 Properties of Incidence Geometries and Coset Geometries . . . . .</b>	<b>4764</b>
CosetGeometry(G, S)	4756	IsFTGeometry(D)	4764
<b>142.3 Elementary Invariants . . .</b>	<b>4759</b>	IsFTGeometry(C)	4764
Points(D)	4759	IsFirm(X)	4764
Elements(D)	4759	IsThin(X)	4765
Types(D)	4759	IsThick(X)	4765
Types(C)	4759	IsResiduallyConnected(X)	4765
Rank(D)	4759	IsRC(X)	4765
Rank(C)	4759	IsGraph(D)	4765
IncidenceGraph(D)	4760	IsGraph(C)	4765
Group(C)	4760	<b>142.11 Intersection Properties of Coset Geometries . . . . .</b>	<b>4765</b>
MaxParabolics(C)	4760	HasIntersectionPropertyN(C, n)	4766
MaximalParabolics(C)	4760	HasIntersectionProperty(C)	4766
MinParabolics(C)	4760	HasWeakIntersectionProperty(C)	4766
MinimalParabolics(C)	4760	<b>142.12 Primitivity Properties on Coset Geometries . . . . .</b>	<b>4766</b>
Borel(C)	4760	IsPrimitive(C)	4766
BorelSubgroup(C)	4760	IsPRI(C)	4766
Kernel(C)	4760	IsWeaklyPrimitive(C)	4766
Kernels(C)	4760	IsWPRI(C)	4766
Quotient(C, K)	4760	IsResiduallyPrimitive(C)	4766
<b>142.4 Conversion Functions . . .</b>	<b>4761</b>	IsRPRI(C)	4766
IncidenceGeometry(C)	4761	IsResiduallyWealyPrimitive(C)	4766
CosetGeometry(D)	4761	IsRWPRI(C)	4766
Graph(D)	4761	IsRWP(C)	4766
Graph(C)	4761	IsLocallyTwoTransitive(C)	4767
<b>142.5 Residues . . . . .</b>	<b>4762</b>	Is2T1(C)	4767
Residue(D, f)	4762	<b>142.13 Diagram of an Incidence Geometry . . . . .</b>	<b>4767</b>
Residue(C, f)	4762	Diagram(D)	4767
<b>142.6 Truncations . . . . .</b>	<b>4763</b>	Diagram(C)	4767
Truncation(D, t)	4763	<b>142.14 Bibliography . . . . .</b>	<b>4770</b>
Truncation(C, t)	4763		
<b>142.7 Shadows . . . . .</b>	<b>4763</b>		
Shadow(D, I, F)	4763		



# Chapter 142

## INCIDENCE GEOMETRY

### 142.1 Introduction

This chapter presents the functions designed for constructing and computing with incidence geometries and coset geometries.

We recall the basic definitions and notation in the field of Incidence Geometry. We refer to the *Handbook of Incidence Geometry* [Bue95], edited by Francis Buekenhout, or to Antonio Pasini's book *Diagram Geometries* [Pas94], for a more detailed overview of the subject.

Let  $X$  and  $I$  be two finite sets. Let  $t : X \rightarrow I$  be a mapping from  $X$  onto  $I$ . Let  $\sim$  be a reflexive and symmetric relation such that  $\forall x, y \in X, x \sim y$  and  $t(x) = t(y) \Rightarrow x = y$ . The four-tuple  $\Gamma(X, \sim, t, I)$  is what we call an *Incidence Geometry* in MAGMA. Remark that it is not a geometry in the sense of Buekenhout since we do not impose that every flag (i.e. clique of the incidence graph) of  $\Gamma$  must be contained in a chamber (i.e. a clique containing one element of each type). If the latter condition is satisfied, then an incidence geometry is a geometry in the sense of Buekenhout. The set  $X$  contains the *elements* of the geometry, while  $I$  is called the set of *types*. The function  $t$  is called the *type function* and  $\sim$  is called the *incidence relation* of  $\Gamma$ . The cardinality of  $I$  is the *rank* of  $\Gamma$ .

It is possible to construct incidence geometries from a group and some of its subgroups using an algorithm first introduced by Jacques Tits in 1962 [Tit62]. Let  $G$  be a group and let  $I$  be a finite set. Let  $\{G_i, i \in I\}$  be a set of subgroups of  $G$ . Define  $X = \{G_i g, g \in G, i \in I\}$  to be the set of elements of  $\Gamma$ . Define the type function as  $t : X \rightarrow I : G_i g \rightarrow i$  and the incidence relation as follows:  $G_i g \sim G_j h$  iff  $G_i g \cap G_j h \neq \emptyset$ . The subgroups  $\{G_i, i \in I\}$  are called the *maximal parabolic subgroups*. The subgroup  $\bigcap_{i \in I} G_i$  is called the *Borel subgroup*. Finally, the subgroups  $\{\bigcap_{j \in I \setminus \{i\}} G_j, i \in I\}$  are called the *minimal parabolic subgroups*. These geometries are called *Coset Geometries* in MAGMA to remind the user that they are constructed from a group. Again, a coset geometry is not a geometry in the sense of Buekenhout. If every flag of the coset geometry is contained in a chamber, then it is a Buekenhout geometry. We will see that, using coset geometries, it is easy to build huge incidence geometries by giving very little data.

The category names for the incidence geometries and coset geometries are:

- Incidence Geometry : `IncGeom`
- Coset Geometry : `CosetGeom`

## 142.2 Construction of Incidence and Coset Geometries

### 142.2.1 Construction of an Incidence Geometry

IncidenceGeometry(G)

Construct the incidence geometry  $IG$  having as incidence graph the (labelled) graph  $G$ .

If  $G$  is an unlabelled graph, then the set of elements of  $IG$  is the set of vertices and edges of  $G$ , the set of types is  $\{@1, 2@\}$  where elements of type 1 are vertices of  $G$  and elements of type 2 are edges of  $G$ . An element  $x$  of type 1 is incident to an element  $y$  of type 2 iff the vertex  $x$  in  $G$  is on the edge  $y$  of  $G$ .

If  $G$  is a labelled graph, i.e. a graph whose vertices have labels, then the set of elements of  $IG$  is the set of vertices of  $G$ , the set of types is the set of labels of the vertices of  $G$  and the incidence graph is  $G$ .

The function returns one value: the incidence geometry  $IG$ .

#### Example H142E1

---

The Petersen graph, considered as a rank two incidence geometry, may be constructed by the following statement:

```
> gr := Graph<10|
>   {1,2},{1,5},{1,6},{2,7},{2,3},{3,8},{3,4},{4,9},{4,5},{5,10},
>   {6,8},{8,10},{10,7},{7,9},{9,6}>;
> ig := IncidenceGeometry(gr);
> ig;
```

Incidence geometry ig with 2 types and with underlying graph:

Graph

Vertex	Neighbours
1	11 12 13 ;
2	11 14 15 ;
3	14 16 17 ;
4	16 18 19 ;
5	12 18 20 ;
6	13 21 22 ;
7	15 23 24 ;
8	17 21 25 ;
9	19 22 23 ;
10	20 24 25 ;
11	1 2 ;
12	1 5 ;
13	1 6 ;
14	2 3 ;
15	2 7 ;
16	3 4 ;
17	3 8 ;
18	4 5 ;

```

19      4 9 ;
20      5 10 ;
21      6 8 ;
22      6 9 ;
23      7 9 ;
24      7 10 ;
25      8 10 ;

```

---

**Example H142E2**

The rank three geometry consisting of the vertices, edges and faces of the cube, with symmetric inclusion as incidence might be encoded in the following way. The first 8 vertices of the graph correspond to the vertices of the cube, the next 12 vertices correspond to the edges of the cube and the last 6 are the faces of the cube.

```

> g := Graph<26|
>   {1,9},{1,12},{1,13},{1,21},{1,22},{1,25},
>   {2,9},{2,10},{2,14},{2,21},{2,22},{2,23},
>   {3,10},{3,11},{3,15},{3,21},{3,23},{3,24},
>   {4,11},{4,12},{4,16},{4,21},{4,24},{4,25},
>   {5,13},{5,18},{5,17},{5,22},{5,25},{5,26},
>   {6,14},{6,19},{6,18},{6,23},{6,22},{6,26},
>   {7,15},{7,19},{7,20},{7,24},{7,26},{7,23},
>   {8,17},{8,20},{8,16},{8,26},{8,24},{8,25},
>   {9,21},{9,22},{10,21},{10,23},{11,21},{11,24},
>   {12,21},{12,25},{13,22},{13,25},{14,22},{14,23},
>   {15,23},{15,24},{16,24},{16,25},{17,25},{17,26},
>   {18,22},{18,26},{19,23},{19,26},{20,24},{20,26}>;
> v := VertexSet(g);
> AssignLabels(v,[1,1,1,1,1,1,1,1,1,2,2,2,2,2,2,2,2,2,2,2,3,3,3,3,3,3]);
> cube := IncidenceGeometry(g);
> cube;

```

Incidence geometry cube with 3 types and with underlying graph:

Graph

Vertex	Neighbours
1	9 12 13 21 22 25 ;
2	9 10 14 21 22 23 ;
3	10 11 15 21 23 24 ;
4	11 12 16 21 24 25 ;
5	13 17 18 22 25 26 ;
6	14 18 19 22 23 26 ;
7	15 19 20 23 24 26 ;
8	16 17 20 24 25 26 ;
9	1 2 21 22 ;
10	2 3 21 23 ;
11	3 4 21 24 ;
12	1 4 21 25 ;
13	1 5 22 25 ;

```

14      2 6 22 23 ;
15      3 7 23 24 ;
16      4 8 24 25 ;
17      5 8 25 26 ;
18      5 6 22 26 ;
19      6 7 23 26 ;
20      7 8 24 26 ;
21      1 2 3 4 9 10 11 12 ;
22      1 2 5 6 9 13 14 18 ;
23      2 3 6 7 10 14 15 19 ;
24      3 4 7 8 11 15 16 20 ;
25      1 4 5 8 12 13 16 17 ;
26      5 6 7 8 17 18 19 20 ;

```

---

**Example H142E3**

The Hoffman-Singleton graph  $HoSi$  : the following description is taken from the book *Diagram Geometry*, by A. Cohen and F. Buekenhout [BCon].

Start with the set  $X_1 = \binom{7}{3}$  of all triples from 7. There are 30 collections of 7 triples that form the lines of a *Fano plane* with point set 7. The group  $Sym(7)$  acts transitively on this set of 30 elements, but the alternating group  $Alt(7)$  has two orbits, of cardinality 15 each. Select one and call it  $X_2$ . Now consider the following graph  $\Omega$  constructed from the incidence graph  $(X_1 \cup X_2, *)$  by additionally joining two elements of  $X_1$  whenever they have empty intersection. The following lines of MAGMA-code implement the Hoffman-Singleton graph as an incidence geometry of rank 2. We will study it more later.

```

> HoffmanSingletonGraph := function()
>   pentagram := Graph< 5 | { {1,3}, {3,5}, {5,2}, {2,4}, {4,1} } >;
>   pentagon := PolygonGraph(5);
>   PP := pentagram join pentagon;
>   HS := &join [ PP : i in [1..5] ];
>   return HS + { { Vertices(HS) | i + j*10, k*10 + 6 + (i+j*k) mod 5 } :
>                 i in [1..5], j in [0..4], k in [0..4] };
> end function;
> ig := IncidenceGeometry(HoffmanSingletonGraph());

```

---

**Example H142E4**

A rank four geometry constructed from the Hoffman-Singleton graph : the following description is taken from the book *Diagram Geometry*, by A. Cohen and F. Buekenhout.

Take  $X_1$  and  $X_{1'}$  to be two copies of the vertex set of  $HoSi$ . The group  $Sym(7)$  acts transitively on the set of maximal cocliques of  $HoSi$ . The subgroup  $Alt(7)$  has two orbits of size 15 each. Let  $X_2$  be one of them and  $X_{2'}$  be a copy of  $X_2$ . Define incidence  $\sim$  for  $a \in X_1$ ,  $a' \in X_{1'}$ ,  $b \in X_2$  and  $b' \in X_{2'}$ , by

$a \sim a' \Leftrightarrow \{a, a'\}$  is an edge of  $HoSi$   
 $a \sim b \Leftrightarrow a \notin b$   
 $a \sim b' \Leftrightarrow a \in b'$   
 $a' \sim b \Leftrightarrow a' \notin b$

$$a' \sim b' \Leftrightarrow a' \notin b'$$

$$b \sim b' \Leftrightarrow b \cap b' = \emptyset$$

The geometry  $\Gamma(X_1 \cup X_{1'} \cup X_2 \cup X_{2'}, \sim)$  is the *Neumaier geometry*. The following lines of MAGMA-code implement this geometry as a rank four incidence geometry. We assume that *ig* is the incidence geometry corresponding to the Hoffman-Singleton graph, as described in the previous example.

```

> gr := HoffmanSingletonGraph();
> ig := IncidenceGeometry(gr);
> aut := AutomorphismGroup(ig);
> n := NormalSubgroups(aut);
> X1 := VertexSet(gr);
> g := CompleteGraph(50);
> e := EdgeSet(gr);
> ebis := [];
> for x in EdgeSet(gr) do
>   for i := 1 to 50 do
>     for j := i+1 to 50 do
>       if VertexSet(gr)!i in x then
>         if VertexSet(gr)!j in x then
>           Append(~ebis,[i,j]);
>         end if;
>       end if;
>     end for;
>   end for;
> end for;
> for i := 1 to #ebis do
>   RemoveEdge(~g,ebis[i][1],ebis[i][2]);
> end for;
> c := MaximumClique(g);
> cbis := [];
> for i := 1 to 50 do
>   if (VertexSet(gr)!i in c) then Append(~cbis,i); end if;
> end for;
> c := Set(cbis);
> X2 := Orbit(n[2]'subgroup,c);
> X2 := SetToSequence(X2);
> e := [];
> for i := 1 to 50 do
>   for j := 1 to 50 do
>     if X1!i in Neighbours(X1!j) then Append(~e,{i,j+50}); end if;
>   end for;
> end for;
> for i := 1 to 50 do
>   for j := 1 to 50 do
>     if not(i in X2[j]) then Append(~e,{i,j+100});end if;
>   end for;
> end for;

```

```

> for i :=1 to 50 do
>   for j := 1 to 50 do
>     if i in X2[j] then Append(~e,{i,j+150});end if;
>   end for;
> end for;
> for i :=1 to 50 do
>   for j := 1 to 50 do
>     if i in X2[j] then Append(~e,{i+50,j+100});end if;
>   end for;
> end for;
> for i :=1 to 50 do
>   for j := 1 to 50 do
>     if not(i in X2[j]) then Append(~e,{i+50,j+150});end if;
>   end for;
> end for;
> for i :=1 to 50 do
>   for j := 1 to 50 do
>     if X2[i] meet X2[j] eq {} then Append(~e,{i+100,j+150});end if;
>   end for;
> end for;
> gr2 := Graph<200|Set(e)>;
> v := VertexSet(gr2);
> labs := [i: j in {1..50}, i in {1..4}];
> AssignLabels(v,labs);
> neumaier := IncidenceGeometry(gr2);

```

## 142.2.2 Construction of a Coset Geometry

CosetGeometry( $G$ ,  $S$ ,  $I$ )

Construct the coset geometry  $CG$  with set of types  $I$ , obtained from the group  $G$  and the set  $S$  of subgroups of  $G$ . The sets  $S$  and  $I$  must have same cardinality.

If  $G$  is a permutation group and  $S$  is a set of subgroups of  $G$ , the corresponding coset geometry, obtained by using Tits' algorithm (see above) is constructed. If  $S$  and  $I$  are indexed sets, then the cosets of the subgroup  $S[i]$  are elements of type  $I[i]$ , for all  $i \in \{0, \dots, n-1\}$  where  $n$  is the cardinality of  $S$  (and  $I$ ). The function returns one value: the coset geometry  $CG$ .

CosetGeometry( $G$ ,  $S$ )

Construct the coset geometry  $CG$  obtained from the group  $G$  and the set  $S$  of subgroups of  $G$ .

If  $G$  is a permutation group and  $S$  is a set of subgroups of  $G$ , the corresponding coset geometry, obtained by using Tits' algorithm (see above) is constructed. To every subgroup in  $S$  is associated a number from 0 to  $n-1$  where  $n$  is the cardinality of  $S$ . The function returns one value: the coset geometry  $CG$ .



**Example H142E5**

---

The Petersen graph, considered as a rank two coset geometry, can be implemented in the following way :

```
> g := Sym(5);
> g0 := Stabilizer(g,{1,2});
> g01 := Stabilizer(g, [{1,2},{3,4}]);
> g1 := sub<g|g01,(1,3)(2,4)>;
> cg := CosetGeometry(g,{g0,g1});
> cg;
Coset geometry cg with 2 types
Group:
Symmetric group g acting on a set of cardinality 5
Order = 120 = 2^3 * 3 * 5
      (1, 2, 3, 4, 5)
      (1, 2)
Maximal Parabolic Subgroups:
Permutation group g0 acting on a set of cardinality 5
Order = 12 = 2^2 * 3
      (3, 4)
      (1, 2)
      (4, 5)
Permutation group g1 acting on a set of cardinality 5
Order = 8 = 2^3
      (1, 2)
      (3, 4)
      (1, 3)(2, 4)
Type Set:
{@ 0, 1 @}
```

**Example H142E6**

---

The rank three geometry consisting of the vertices, edges and faces of the cube, can be implemented as a coset geometry in the following way, which is much easier than having to give the full incidence graph :

```
> g := sub<Sym(8)|
>      (1,2,3,4)(5,6,7,8), (1,5)(2,6)(3,7)(4,8), (2,4,5)(3,8,6)>;
> g0 := Stabilizer(g,1);
> g1 := Stabilizer(g,{1,2});
> g2 := Stabilizer(g,{1,2,3,4});
> cg := CosetGeometry(g,{g0,g1,g2});
> cg;
Coset geometry cg with 3 types
Group:
Permutation group g acting on a set of cardinality 8
Order = 48 = 2^4 * 3
      (1, 2, 3, 4)(5, 6, 7, 8)
```

```

      (1, 5)(2, 6)(3, 7)(4, 8)
      (2, 4, 5)(3, 8, 6)
Maximal Parabolic Subgroups:
Permutation group acting on a set of cardinality 8
Order = 8 = 2^3
      (1, 2)(3, 4)(5, 6)(7, 8)
      (2, 4)(6, 8)
Permutation group acting on a set of cardinality 8
Order = 6 = 2 * 3
      (2, 4, 5)(3, 8, 6)
      (3, 6)(4, 5)
Permutation group acting on a set of cardinality 8
Order = 4 = 2^2
      (1, 2)(3, 5)(4, 6)(7, 8)
      (1, 2)(3, 4)(5, 6)(7, 8)
Type Set:
{@ 0, 1, 2 @}

```

### Example H142E7

---

The following lines of MAGMA-code construct a coset geometry of rank 6 for the group  $Sym(6)$ .

```

> g := Sym(6);
> s := [Stabilizer(g,{1..j}) : j in {1..5}];
> cg := CosetGeometry(g,Set(s));
> cg;
Coset geometry cg with 5 types
Group:
Symmetric group g acting on a set of cardinality 6
Order = 720 = 2^4 * 3^2 * 5
      (1, 2, 3, 4, 5, 6)
      (1, 2)
Maximal Parabolic Subgroups:
Permutation group acting on a set of cardinality 6
Order = 48 = 2^4 * 3
      (3, 4)
      (4, 5)
      (5, 6)
      (1, 2)
Permutation group acting on a set of cardinality 6
Order = 48 = 2^4 * 3
      (1, 2)
      (2, 3)
      (3, 4)
      (5, 6)
Permutation group acting on a set of cardinality 6
Order = 36 = 2^2 * 3^2
      (1, 2)

```

```

(4, 5)
(5, 6)
(2, 3)
Permutation group acting on a set of cardinality 6
Order = 120 = 2^3 * 3 * 5
(2, 3)
(3, 4)
(4, 5)
(5, 6)
Permutation group acting on a set of cardinality 6
Order = 120 = 2^3 * 3 * 5
(1, 2)
(2, 3)
(3, 4)
(4, 5)
Type Set:
{@ 0, 1, 2, 3, 4 @}

```

---

### 142.3 Elementary Invariants

**Points(D)**

**Elements(D)**

Given an incidence geometry  $D$ , return the set of elements of  $D$ . These elements are the points of the incidence graph of  $D$ .

**Types(D)**

Given an incidence geometry  $D$ , return the set of types of  $D$ .

**Types(C)**

Given a coset geometry  $C$ , return the set of types of  $C$ .

**Rank(D)**

Given an incidence geometry  $D$ , return the rank of  $D$ , i.e. the cardinality of the set of types.

**Rank(C)**

Given a coset geometry  $C$ , return the rank of  $C$ .

**IncidenceGraph(D)**

Given an incidence geometry  $D$ , return the incidence graph of  $D$ , its vertex set and its edge set.

We remark that this function is not implemented for coset geometries but we may convert a coset geometry into an incidence geometry using **IncidenceGeometry** and then compute its incidence graph.

**Group(C)**

Given a coset geometry  $C$ , return the group from which  $C$  is constructed.

**MaxParabolics(C)****MaximalParabolics(C)**

Given a coset geometry  $C$ , return an indexed set containing the maximal parabolics of  $C$ .

**MinParabolics(C)****MinimalParabolics(C)**

Given a coset geometry  $C$ , return an indexed set containing the minimal parabolics of  $C$ .

**Borel(C)****BorelSubgroup(C)**

Given a coset geometry  $C$ , return the *Borel subgroup* of  $C$ , i.e. the intersection of all maximal parabolic subgroups of  $C$ .

**Kernel(C)**

Given a coset geometry  $C$ , return a permutation group which is its kernel, i.e. the subgroup of the Borel subgroup of  $C$  that fixes all elements of the geometry  $C$ .

**Kernels(C)**

Given a coset geometry  $C$ , return a sequence containing the  $i$ -kernel  $K_i$  of each maximal parabolic subgroup  $G_i$  of  $C$ . The  $i$ -kernel of the subgroup  $G_i$  is the subgroup consisting of all the elements of  $G_i$  that fix all the elements of the residue of  $G_i$ .

**Quotient(C, K)**

Given a coset geometry  $C = (G; (G_i)_{i \in I})$  and a permutation group  $K$ , return the coset geometry  $(G/K; (G_i/K)_{i \in I})$  provided that  $K$  is a normal subgroup of  $G$  and of all the maximal parabolic subgroups of  $C$ .

## 142.4 Conversion Functions

In this section we describe the functions that are available to convert incidence geometries and coset geometries in other objects.

### **IncidenceGeometry(C)**

Construct the incidence geometry  $IG$  from the coset geometry  $C$ . This is done using Tits' algorithm described in the introduction of this chapter. The function returns one value: the incidence geometry  $IG$ .

### **CosetGeometry(D)**

Convert the incidence geometry  $D$  into a coset geometry.

If  $D$  is an incidence geometry that can be converted into a coset geometry, the coset geometry isomorphic to it is constructed in the following way. The group  $G$  of the coset geometry  $CG$  is the automorphism group of  $D$ . MAGMA determines a chamber  $C$  of  $D$ , that is a clique of the incidence graph of  $D$  containing one element of each type. To every element  $x$  in  $C$ , MAGMA associates a subgroup  $G_x$  which is the stabilizer of  $x$  in  $G$ . The subgroups  $(G_x, x \in C)$  are the maximal parabolic subgroups of  $CG$ . In order to obtain a coset geometry combinatorially isomorphic to the incidence geometry we started with, the group  $G$  must be transitive on every rank two truncation of  $D$ . If this condition is satisfied, the function returns a boolean set to the value **true** and the coset geometry  $CG$ . Otherwise, the function returns **false**.

### **Graph(D)**

If **IsGraph(D)** returns **true**, this function constructs the undirected graph corresponding to the incidence geometry  $D$ .

### **Graph(C)**

If **IsGraph(C)** returns **true**, this function constructs the undirected graph corresponding to the coset geometry  $C$ .

### Example H142E8

---

Taking back the last example for incidence geometries, we can convert the Neumaier geometry into a coset geometry by typing the following command (**neumaier** is the Neumaier geometry constructed above):

```
> ok,cg := CosetGeometry(neumaier);
> ok;
true
```

This means the conversion has been done successfully. So  $cg$  is the coset geometry corresponding to **neumaier**.

```
> cg;
Coset geometry cg with 4 types
Group:
```

Permutation group acting on a set of cardinality 200

Order = 126000 =  $2^4 * 3^2 * 5^3 * 7$

Maximal Parabolic Subgroups:

Permutation group acting on a set of cardinality 200

Order = 2520 =  $2^3 * 3^2 * 5 * 7$

Permutation group acting on a set of cardinality 200

Order = 2520 =  $2^3 * 3^2 * 5 * 7$

Permutation group acting on a set of cardinality 200

Order = 2520 =  $2^3 * 3^2 * 5 * 7$

Permutation group acting on a set of cardinality 200

Order = 2520 =  $2^3 * 3^2 * 5 * 7$

Type Set:

{@ 1, 2, 3, 4 @}

## 142.5 Residues

Let  $\Gamma(X, \sim, t, I)$  be an incidence geometry and let  $F$  be a flag of  $\Gamma$  (i.e. a clique of the incidence graph of  $\Gamma$ ).

We say that an element  $x \in F$  is incident to the flag  $F$  if and only if  $x$  is incident to all elements in  $F$ , and we denote it  $x \sim F$ .

The *residue*  $\Gamma_F$  of the flag  $F$  in  $\Gamma$  is the geometry whose set of elements is  $\{x \in X : x \sim F\} \setminus F$  and whose set of types is  $I \setminus t(F)$ , together with the restricted type function and incidence relation.

Let  $\Gamma(G; (G_i)_{i \in I})$  be a coset geometry and assume that  $G$  acts flag-transitively on  $\Gamma$ . Let  $F$  be a flag of  $\Gamma$ . The *residue* of  $F$  is the coset geometry  $\Gamma_F = \Gamma(\cap_{j \in F} G_j; (G_i \cap (\cap_{j \in F} G_j))_{i \in I \setminus t(F)})$ .

Residue(D, f)
---------------

Given an incidence geometry  $D$  and a flag  $f$  of  $D$ , return the residue of the flag  $f$  as an incidence geometry.

Residue(C, f)
---------------

Given a coset geometry  $C$  and a subset  $f$  of the set of types of  $C$ , return the residue of the flag consisting in the maximal parabolics of  $C$  whose type is in  $f$ .

### 142.6 Truncations

Let  $\Gamma(X, \sim, t, I)$  be an incidence geometry and let  $J$  be a subset of  $I$ . Then the  $J$ -truncation of  $\Gamma$  is the geometry whose set of elements is  $t^{-1}(J)$ , together with the restricted type function and incidence relation.

Let  $J \subseteq I$ . The  $J$ -truncation of the coset geometry  $\Gamma(G; (G_i)_{i \in I})$  is the coset geometry  $\Gamma(G; (G_j)_{j \in J})$ .

<b>Truncation(<math>D, t</math>)</b>
--------------------------------------

Given an incidence geometry  $D$  and  $t$  a subset of the set of types of  $D$ , return the  $t$ -truncation of  $D$  as an incidence geometry.

<b>Truncation(<math>C, t</math>)</b>
--------------------------------------

Given a coset geometry  $C$  and  $t$  a subset of the set of types of  $C$ , return the  $t$ -truncation of  $C$  as a coset geometry.

### 142.7 Shadows

Let  $\Gamma(X, \sim, t, I)$  be an incidence geometry. Let  $J$  be a subset of  $I$  and let  $F$  be a flag of  $\Gamma$  such that  $t(F) \cap J = \emptyset$ . The  $J$ -shadow of the flag  $F$ , denoted by  $\sigma_J(F)$ , is the set of flags of type  $J$  in the residue of the flag  $F$ .

<b>Shadow(<math>D, I, F</math>)</b>
-------------------------------------

Given an incidence geometry  $D$ , a subset  $I$  of the set of types of  $D$  and a flag  $F$  of  $D$ , return the  $I$ -shadow of the flag  $F$  as an indexed set of subsets of points of  $D$ .

### 142.8 Shadow Spaces

Let  $\Gamma(X, \sim, t, I)$  be an incidence geometry. Let  $J$  be a subset of  $I$ . The *shadow space*  $\Gamma(J)$  of  $\Gamma$  is the incidence structure whose point-set is the set of flags of type  $J$  of  $\Gamma$  and whose blocks are the  $J$ -shadows of the flags of  $\Gamma$ .

<b>ShadowSpace(<math>D, I</math>)</b>
---------------------------------------

Given an incidence geometry  $D$  and a subset  $I$  of the set of types of  $D$ , return the shadow space  $D(I)$  as an incidence structure.

### 142.9 Automorphism Group and Correlation Group

These function are currently only available for incidence geometries.

An *automorphism*  $\alpha$  of an incidence geometry  $\Gamma(X, \sim, t, I)$  is an automorphism of the incidence graph of  $\Gamma$  such that for all  $x \in X$ ,  $t(\alpha(x)) = t(x)$ . In other words, an automorphism cannot change the type of an element. The *automorphism group* of  $\Gamma$ , denoted  $Aut(\Gamma)$ , is the group of all automorphisms of  $\Gamma$ .

A *correlation*  $\alpha$  of an incidence geometry  $\Gamma(X, \sim, t, I)$  is an automorphism of the incidence graph of  $\Gamma$  such that for all  $x, y \in X$ ,  $t(x) = t(y) \Rightarrow t(\alpha(x)) = t(\alpha(y))$ . The *correlation group* of  $\Gamma$ , denoted  $Cor(\Gamma)$ , is the group of all correlations of  $\Gamma$ .

It is obvious that  $Aut(\Gamma)$  is a subgroup of  $Cor(\Gamma)$ .

For an incidence geometry  $\Gamma$ , we can compute  $Aut(\Gamma)$  and  $Cor(\Gamma)$  using the commands described below.

**AutomorphismGroup(D)**

Given an incidence geometry  $D$ , return the group of type-preserving automorphisms of  $D$  as a permutation group of type **GrpPerm** acting on the set of elements of  $D$ .

**CorrelationGroup(D)**

Given an incidence geometry  $D$ , return the group of automorphisms of  $D$  as a permutation group of type **GrpPerm** acting of the set on elements of  $D$ .

### 142.10 Properties of Incidence Geometries and Coset Geometries

Let us recall definitions of the properties described in this section.

An incidence geometry  $\Gamma$  is *flag-transitive* if for every two flags  $x, y$  of the same type of  $\Gamma$ , there exists an element  $g$  of  $Aut(\Gamma)$  such that  $g(x) = y$ . We also say that  $Aut(\Gamma)$  acts flag-transitively in this case.

Moreover, it is a flag-transitive geometry if it contains at least one chamber.

A coset geometry  $\Gamma(G; (G_i)_{i \in I})$  is *flag-transitive* if for every two flags  $x, y$  of the same type of  $\Gamma$ , there exists an element  $g$  of  $G$  such that  $g(x) = y$ . It is then a flag-transitive geometry since the set  $\{(G_i)_{i \in I}\}$  is a chamber of  $\Gamma$ .

**IsFTGeometry(D)**

Given an incidence geometry  $D$ , return **true** if and only if the automorphism group of  $D$  acts flag-transitively on  $D$  and  $D$  has at least one chamber.

**IsFTGeometry(C)**

Given a coset geometry  $C$ , return **true** if and only if the group of  $C$  acts flag-transitively on  $C$ .

**IsFirm(X)**

Given either a coset geometry or an incidence geometry  $X$  that is flag transitive, return **true** if and only if every flag of  $X$  is contained in at least two chambers.



**IsThin(X)**

Given either a coset geometry or an incidence geometry  $X$  that is flag transitive, return **true** if and only if every flag of  $X$  is contained in exactly two chambers.

**IsThick(X)**

Given either a coset geometry or an incidence geometry  $X$  that is flag transitive, return **true** if and only if every flag of the geometry is contained in exactly three chambers.

**IsResiduallyConnected(X)****IsRC(X)**

Given either a coset geometry or an incidence geometry  $X$  that is flag transitive, return **true** if and only if every residue of rank at least two of  $X$  has a connected incidence graph.

**IsGraph(D)**

Given an incidence geometry  $D$ , tests if this incidence geometry corresponds to a graph:  $D$  must be of rank two and such that for one of the two types, say  $e$ , all elements of this type are incident with exactly two elements of the other type. Elements of type  $e$  then correspond to edges of an undirected graph and elements of the other type to the vertices of that graph.

**IsGraph(C)**

Given a coset geometry  $C$ , tests if this geometry corresponds to a graph:  $C$  must be of rank two and one of the two maximal parabolic subgroups, say  $G_e$ , must contain the Borel subgroup as a subgroup of index 2. In that case, the cosets of  $G_e$  correspond to edges of a graph and the cosets of the other maximal parabolic subgroup correspond to the vertices of this graph.

## 142.11 Intersection Properties of Coset Geometries

Let  $\Gamma(X, *, t, I)$  be an incidence geometry. Given a type  $i \in I$ , for any flag  $F$  of  $\Gamma$ , we define the  $i$ -shadow  $\sigma_i(F)$  as the set of elements of type  $i$  incident with  $F$ .

We define the intersection property  $(IP)$  as it appears in [Bue79].

*(IP) for every type  $i$ , the intersection of the  $i$ -shadows of a variety  $x$  and a flag  $F$  is empty or it is the  $i$ -shadow of a flag incident to  $x$  and  $F$ . The same holds on the residues.*

In earlier works, the second author, together with other persons, among whom are Francis Buekenhout, Michel Dehon and Philippe Cara, imposed a condition called  $(IP)_2$ . This condition asks that all rank two residues of  $\Gamma$  satisfy  $(IP)$ .

If  $\Gamma$  is a geometry of rank  $n$ , we could define a property  $(IP)_k$  in the following way (for  $k = 2, \dots, n$ ) as suggested by Francis Buekenhout.

*$(IP)_k$  for every residue  $R$  of rank  $k$  of  $\Gamma$ , for every type  $i$  in the set of types of  $R$ , the intersection of the  $i$ -shadows of a variety  $x$  and a flag  $F$  is empty or it is the  $i$ -shadow of a flag incident to  $x$  and  $F$ .*

In [JL04], Pascale Jacobs and Dimitri Leemans have designed good algorithms to test these intersection properties. The Magma implementation is available with the following functions.

**HasIntersectionPropertyN(C,n)**

Given a coset geometry  $C$  and a positive integer  $n$ , this function determines firstly, whether  $C$  satisfies the intersection property of rank  $n$ , and secondly, whether  $C$  satisfies the weak intersection property of rank  $n$ . A boolean value corresponding to each of these cases is returned.

**HasIntersectionProperty(C)**

Given a coset geometry  $C$ , this function returns the boolean value **true** if and only if  $C$  satisfies the intersection property.

**HasWeakIntersectionProperty(C)**

Given a coset geometry  $C$ , this function returns the boolean value **true** if and only if  $C$  satisfies the weak intersection property.

## 142.12 Primitivity Properties on Coset Geometries

**IsPrimitive(C)**

**IsPRI(C)**

Given a coset geometry  $C$ , this function returns the boolean value **true** if and only if  $C$  is a primitive geometry, i.e. all of its maximal parabolic subgroups are maximal subgroups of its group.

**IsWeaklyPrimitive(C)**

**IsWPRI(C)**

Given a coset geometry  $C$ , this function returns the boolean value **true** if and only if  $C$  is a weakly primitive geometry, i.e. at least one of its maximal parabolic subgroups is a maximal subgroup of its group.

**IsResiduallyPrimitive(C)**

**IsRPRI(C)**

Given a coset geometry  $C$ , this function returns the boolean value **true** if and only if  $C$  is a primitive geometry and all of its residues are as well.

**IsResiduallyWeaklyPrimitive(C)**

**IsRWPRI(C)**

**IsRWP(C)**

Given a coset geometry  $C$ , this function returns the boolean value **true** if and only if  $C$  is a weakly primitive geometry and all of its residues are as well.

IsLocallyTwoTransitive(C)
---------------------------

Is2T1(C)
----------

Given a coset geometry  $C$ , this function returns the boolean value **true** if and only if  $C$  is locally two-transitive, i.e. all of its minimal parabolic subgroups have a two-transitive action on the cosets of the Borel subgroup.

### 142.13 Diagram of an Incidence Geometry

As Francis Buekenhout defined it in [Bue79], the diagram of a firm, residually connected, flag-transitive geometry  $\Gamma$  is a complete graph  $K$ , whose vertices are the elements of the set of types  $I$  of  $\Gamma$ , provided with some additional structure which is further described as follows. To each vertex  $i \in I$ , we attach the order  $s_i$  which is  $|\Gamma_F| - 1$  where  $F$  is any flag of type  $I \setminus \{i\}$ , and the number  $n_i$  of elements of type  $i$  of  $\Gamma$ . To every edge  $\{i, j\}$  of  $K$ , we associate three positive integers  $d_{ij}$ ,  $g_{ij}$  and  $d_{ji}$  where  $g_{ij}$  (the gonality) is equal to half the girth of the incidence graph of a residue  $\Gamma_F$  of type  $\{i, j\}$ , and  $d_{ij}$  (resp.  $d_{ji}$ ), the  $i$ -diameter (resp.  $j$ -diameter) is the greatest distance from some fixed  $i$ -element (resp.  $j$ -element) to any other element in  $\Gamma_F$ .

On a picture of the diagram, this structure will often be depicted as follows.

```

i  d_ij  g_ij  d_ji  j
0-----0
s_i-1          s_j-1
n_i           n_j

```

Moreover, when  $g_{ij} = d_{ij} = d_{ji} = n \neq 2$ , we only write  $g_{ij}$  and if  $n = 2$ , we do not draw the edge  $\{i, j\}$ .

Diagram(D)
------------

Diagram(C)
------------

Given a firm, residually connected and flag-transitive incidence geometry  $D$  or a coset geometry  $C$ , return a complete graph  $K$  whose vertices and edges are labelled in the following way. Every vertex  $i$  of  $K$  is labelled with a sequence of two positive integers, the first one being  $s_i$  and the second one being the number of elements of type  $i$  in  $D$ . Every edge  $\{i, j\}$  is labelled with a sequence of three positive integers which are respectively  $d_{ij}$ ,  $g_{ij}$  and  $d_{ji}$ .

Observe that both these functions do the same but the second one works much faster than the first one since it uses groups to compute the parameters of the diagram. So when the user has to compute the diagram of an incidence geometry, it is strongly advised that he first converts it into a coset geometry and then computes the diagram of the corresponding coset geometry.

**Example H142E9**

---

Back to the Petersen graph : let us now explore a little more the first incidence geometry we constructed. Suppose that  $ig$  is the rank two incidence geometry corresponding to the Petersen graph. We may test whether it is firm, residually connected and flag-transitive.

```
> IsFirm(ig);
true
> IsRC(ig);
true
> IsFTGeometry(ig);
true
```

Since it satisfies these three properties, we can compute its diagram.

```
> d, v, e := Diagram(ig);
> d;
Graph
Vertex Neighbours
1      2 ;
2      1 ;
> for x in v do print x, Label(x);end for;
1 [ 1, 10 ]
2 [ 2, 15 ]
> for x in e do print x, Label(x);end for;
{1, 2} [ 5, 5, 6 ]
```

We thus see that the diagram of  $ig$  can be drawn as follows.

```
1      5 5 6      2
0-----0
1      2
10     15
```

**Example H142E10**

---

Back to the cube: let  $ig$  be the rank three incidence geometry of the cube as constructed above and compute its diagram.

```
> cube := IncidenceGeometry(g);
> d, v, e := Diagram(cube);
> d;
Graph
Vertex Neighbours
1      2 3 ;
2      1 3 ;
3      1 2 ;
> for x in v do x, Label(x); end for;
1 [ 1, 8 ]
2 [ 1, 12 ]
3 [ 1, 6 ]
```

```
> for x in e do x, Label(x); end for;
{1, 2} [ 4, 4, 4 ]
{1, 3} [ 2, 2, 2 ]
{2, 3} [ 3, 3, 3 ]
```

So the diagram can be depicted as follows.

1	4	2	3	3
0-----0-----0				
1		1		1
8		12		6

---

### Example H142E11

Back to the Hoffman-Singleton graph:

```
> ig := IncidenceGeometry(HoffmanSingletonGraph());
> d, v, e := Diagram(ig); d;
Graph
Vertex Neighbours
1      2 ;
2      1 ;
> for x in v do print x, Label(x); end for;
1 [ 1, 50 ]
2 [ 6, 175 ]
> for x in e do print x, Label(x); end for;
{1, 2} [ 5, 5, 6 ]
```

So the diagram can be depicted as follows.

1	5	5	6	2
0-----0				
1				6
50				175

---

### Example H142E12

Back to Neumaier's geometry : let *ig* be the rank four incidence geometry of Neumaier, as described above and compute its diagram.

```
> d, v, e := Diagram(neumaier);
> d;
Graph
Vertex Neighbours
1      2 3 4 ;
2      1 3 4 ;
3      1 2 4 ;
4      1 2 3 ;
> for x in v do print x, Label(x); end for;
1 [ 2, 50 ]
```

```

2 [ 2, 50 ]
3 [ 2, 50 ]
4 [ 2, 50 ]
> for x in e do print x, Label(x); end for;
{1, 2} [ 4, 4, 4 ]
{1, 3} [ 2, 2, 2 ]
{1, 4} [ 3, 3, 3 ]
{2, 3} [ 3, 3, 3 ]
{2, 4} [ 2, 2, 2 ]
{3, 4} [ 4, 4, 4 ]

```

So the diagram can be depicted as follows.

```

.      1      4      2
.  2  0-----0 2
.  50 |                | 50
.      |                |
.      |                |
.  3 |                | 3
.      |                |
.      |                |
.      |      4      |
.  3 0-----0 4
.      2      2
.  50      50

```

---

## 142.14 Bibliography

- [BCon] Francis Buekenhout and Arjeh M. Cohen. *Diagram geometry*. In preparation.
- [Bue79] Francis Buekenhout. Diagrams for geometries and groups. *J. Combin. Theory Ser. A*, 27(2):121–151, 1979.
- [Bue95] Francis Buekenhout. *Handbook of incidence geometry*. North-Holland, Amsterdam, 1995.
- [JL04] Pascale Jacobs and Dimitri Leemans. An algorithmic analysis of the intersection property. *LMS J. Comput. Math.*, 7:284–299 (electronic), 2004.
- [Pas94] Antonio Pasini. *Diagram geometries*. Oxford Science Publications. The Clarendon Press Oxford University Press, New York, 1994.
- [Tit62] Jacques Tits. Géométries polyédriques et groupes simples. *Atti 2a Riunione Groupem. Math. Express. Lat. Firenze*, pages 66–88, 1962.

# 143 CONVEX POLYTOPES AND POLYHEDRA

## 143.1 Introduction and First Examples . . . . . 4773

## 143.2 Polytopes, Cones and Polyhedra . . . . . 4778

### 143.2.1 Polytopes . . . . . 4778

Polytope(Q)	4778
RandomPolytope(L,n,k)	4778
RandomPolytope(d,n,k)	4778
Polar(P)	4778
CrossPolytope(L)	4778
CrossPolytope(d)	4778
StandardSimplex(L)	4778
StandardSimplex(d)	4778
CyclicPolytope(L,n)	4779
CyclicPolytope(d,n)	4779

### 143.2.2 Cones . . . . . 4779

Cone(A)	4779
Cone(v)	4779
ConeWithInequalities(B)	4779
FullCone(L)	4779
FullCone(n)	4779
PositiveQuadrant(L)	4779
PositiveQuadrant(n)	4779
ZeroCone(L)	4779
ZeroCone(n)	4779
Dual(C)	4780
NormalisedCone(P)	4780
ConeInSublattice(C)	4780
ConeQuotientByLinearSubspace(C)	4780

### 143.2.3 Polyhedra . . . . . 4780

Polyhedron(C,H,h)	4780
Polyhedron(C,H,h)	4780
Polyhedron(C)	4780
HalfspaceToPolyhedron(v,h)	4780
HalfspaceToPolyhedron(Q,h)	4780
HyperplaneToPolyhedron(v,h)	4780
HyperplaneToPolyhedron(Q,h)	4780
Polyhedron(C,f,v)	4781
EmptyPolyhedron(L)	4781
ConeToPolyhedron(C)	4781
PolyhedronInSublattice(P)	4781
FixedSubspaceToPolyhedron(G)	4781
FixedSubspaceToPolyhedron(L,G)	4781

### 143.2.4 Arithmetic Operations on Polyhedra . . . . . 4782

eq	4782
eq	4782
meet	4782
meet	4782
subset	4782
+	4782
+	4782

+	4782
+	4782
*	4782
*	4783
-	4783

## 143.3 Basic Combinatorics of Polytopes and Polyhedra . . . . . 4783

### 143.3.1 Vertices and Inequalities . . . . . 4783

Vertices(P)	4783
NumberOfVertices(P)	4783
Rays(C)	4783
Ray(C,i)	4783
LinearSpanEquations(C)	4783
LinearSpanEquations(Q)	4783
LinearSpanGenerators(C)	4783
LinearSpanGenerators(Q)	4783
LinearSubspaceGenerators(C)	4783
Inequalities(C)	4784
Inequalities(P)	4784
MinimalInequalities(C)	4784

### 143.3.2 Facets and Faces . . . . . 4785

#fVector(C)	4785
fVector(P)	4785
#hVector(C)	4785
#hVector(P)	4785
Facets(C)	4785
Facets(P)	4785
FacetIndices(P)	4785
NumberOfFacets(P)	4785
Faces(C)	4785
Faces(P)	4785
Faces(C,i)	4785
Faces(P,i)	4785
FaceIndices(P,i)	4785
Edges(P)	4785
EdgeIndices(P)	4786
Graph(P)	4786
FaceSupportedBy(C,H)	4786
IsSupportingHyperplane(v,h,P)	4786
SupportingCone(P,v)	4786

## 143.4 The Combinatorics of Polytopes . . . . . 4786

### 143.4.1 Points in Polytopes . . . . . 4786

Points(P)	4786
InteriorPoints(P)	4786
BoundaryPoints(P)	4786
NumberOfPoints(P)	4786

### 143.4.2 Ehrhart Theory of Polytopes . . . 4787

EhrhartSeries(P)	4787
EhrhartPolynomial(P)	4787
EhrhartCoefficients(P,l)	4787
EhrhartCoefficient(P,k)	4787

<i>143.4.3 Automorphisms of a Polytope . . .</i>	<i>4787</i>	.	4795
AutomorphismGroup(P)	4787	Basis(L,i)	4795
<i>143.4.4 Operations on Polytopes . . . .</i>	<i>4788</i>	Basis(L)	4796
Triangulation(P)	4788	Form(L,Q)	4796
TriangulationOfBoundary(P)	4788	Zero(L)	4796
<b>143.5 Cones and Polyhedra . . .</b>	<b>4788</b>	+	4796
<i>143.5.1 Generators of Cones . . . . .</i>	<i>4788</i>	-	4796
ZGenerators(C)	4788	*	4796
RGenerators(C)	4788	/	4796
MinimalRGenerators(C)	4788	eq	4796
Points(C,H,h)	4788	AreProportional(P,Q)	4796
<i>143.5.2 Properties of Polyhedra . . . .</i>	<i>4789</i>	/	4796
CompactPart(P)	4789	in	4797
IntegralPart(P)	4789	IsZero(v)	4797
InfinitePart(P)	4790	IsIntegral(v)	4797
IsEmpty(P)	4790	IsPrimitive(v)	4797
IsMaximalDimension(C)	4792	PrimitiveLatticeVector(v)	4797
IsMaximalDimension(P)	4792	<i>143.6.3 Operations on Toric Lattices . .</i>	<i>4798</i>
IsStrictlyConvex(C)	4792	eq	4798
IsSimplicial(C)	4792	Sublattice(Q)	4798
IsSimplicial(P)	4792	ToricLattice(Q)	4798
IsSimplex(P)	4792	Quotient(C)	4798
IsSimple(P)	4792	Quotient(Q)	4798
IsAffineLinear(P)	4792	Quotient(v)	4798
IsZero(C)	4792	AddVectorToLattice(v)	4798
<i>143.5.3 Attributes of Polyhedra . . . .</i>	<i>4793</i>	AddVectorToLattice(Q)	4798
Dimension(C)	4793	IsSublattice(L)	4798
Dimension(P)	4793	IsSuperlattice(L)	4798
Index(C)	4793	IsDirectSum(L)	4798
IsShellable(P)	4793	IsQuotient(L)	4799
<i>143.5.4 Combinatorics of Polyhedral Com-</i>	<i>4793</i>	Sublattice(L)	4799
<i>plexes . . . . .</i>	<i>4793</i>	Superlattice(L)	4799
Ambient(C)	4793	Summands(L)	4799
Ambient(P)	4793	<i>143.6.4 Maps of Toric Lattices . . . . .</i>	<i>4800</i>
ChangeAmbient(C,L)	4793	ZeroMap(L,K)	4800
ChangeAmbient(P,L)	4793	IdentityMap(L)	4800
<b>143.6 Toric Lattices . . . . .</b>	<b>4793</b>	hom< >	4800
<i>143.6.1 Toric Lattices . . . . .</i>	<i>4794</i>	LatticeMap(L,K,M)	4800
ToricLattice(n)	4794	LatticeMap(L,Q)	4800
ScalarLattice()	4794	DefiningMatrix(f)	4800
Dual(L)	4794	Image(f,C)	4800
+	4795	Image(f,P)	4800
DirectSum(L,M)	4795	Image(f,v)	4800
DirectSum(Q)	4795	Preimage(f,C)	4800
^	4795	Preimage(f,P)	4800
Dimension(L)	4795	Preimage(f,v)	4800
<i>143.6.2 Points of Toric Lattices . . . .</i>	<i>4795</i>	KernelEmbedding(f)	4801
!	4795	KernelEmbedding(v)	4801
LatticeVector(L,Q)	4795	KernelBasis(f)	4801
LatticeVector(Q)	4795	KernelBasis(v)	4801
		ImageBasis(f)	4801
		IsCokernelTorsionFree(f)	4801
		<b>143.7 Bibliography . . . . .</b>	<b>4801</b>



# Chapter 143

## CONVEX POLYTOPES AND POLYHEDRA

### 143.1 Introduction and First Examples

This is a package with tools to build and analyse finite-dimensional, finitely-generated, rational polyhedra. In general a polyhedron is the Minkowski sum of a rational polytope (the compact, convex hull of finitely many rational points) and a cone (the convex hull of finitely many rational half-lines emanating from the origin).

It is important to note that we are discussing only *rational* polyhedra: their vertices lie in some finite-dimensional rational vector space  $L = \mathbf{Q}^n$  with the supporting hyperplanes represented in its dual (loosely speaking, all faces have rational gradients). There is a natural lattice in this setup: the integral sublattice  $\mathbf{Z}^n$  of  $L$ . This is a lattice only in the sense of being a  $\mathbf{Z}$ -module; it does not come with a preferred definite quadratic form (or norm), and so, in particular, lengths of vectors are not defined. (We explain below how volumes are nevertheless defined relative to the integral lattice.) A point of  $L$  is called ‘integral’ if it lies in the integral sublattice, but there is no requirement that vertices of polyhedra be integral. Notwithstanding all these remarks, we refer to such ambient spaces  $L$  as ‘lattices’.

This chapter is organised so that functions applying to compact polyhedra, namely polytopes, are collected together in coherent blocks as much as possible. Section 143.2 lists various ways of constructing polytopes, cones and polyhedra, including Minkowski sum and other arithmetical and set-theoretical operations. Section 117.10.2 presents the basic combinatorics associated to all polyhedra such as recovering their vertices or faces. The next section, Section 117.10.2, is dedicated to polytopes. Most of the functions described here rely on compactness; they include enumeration of points and triangulation. Then Section 117.10.2 details functions that apply to polyhedra more generally, although this includes some functions that apply to cones in particular: computing the integral generators of  $C \cap \mathbf{Z}^n$  as a semigroup, for example, where  $C$  is a cone in  $L$ . Finally, there is section outlining the functions that operate on the ambient lattices. These functions are not usually required, since for the most part the ambient space is a book-keeping device that the package handles automatically, but they arise as domains and codomains of maps. This provides a mechanism for modifying the integral sublattice, which can be used to change the underlying notion of which points count as integral.

Before starting, we outline the capabilities of the package by extended examples. The first illustrates the basic machinery available for the analysis of polytopes—that is, the compact convex hulls of finitely many points.

---

#### Example H143E1

A polytope can be constructed as the convex hull of finitely many points; the points can be denoted simply as sequences of integers (or even rational numbers), although they will be cast as

points of an ambient lattice (that is, a  $\mathbf{Q}$ -vector space marked with a spanning lattice  $\mathbf{Z}^n$ ). For example, we build a polytope  $P$  as the convex hull of the four points  $(1, 0, 0)$ ,  $(0, 1, 0)$ ,  $(1, -3, 5)$  and  $(-2, 2, -5)$ .

```
> P := Polytope([ [0,0,0], [1,0,0], [0,1,0], [1,-3,5], [-2,2,-5] ]);
> P;
3-dimensional polytope P with 5 generators:
  ( 0,  0,  0),
  ( 1,  0,  0),
  ( 0,  1,  0),
  ( 1, -3,  5),
  (-2,  2, -5)
```

No analysis of  $P$  has been carried out in its construction. In fact, one of its defining ‘generators’ is not required, as can be seen from the vertices of  $P$  (which after this calculation will set as the default printing information).

```
> Vertices(P);
[
  (1, 0, 0),
  (0, 1, 0),
  (1, -3, 5),
  (-2, 2, -5)
]
> P;
3-dimensional polytope P with 4 vertices:
  ( 1,  0,  0),
  ( 0,  1,  0),
  ( 1, -3,  5),
  (-2,  2, -5)
```

One can extract the combinatorial components of  $P$ . Here we recover the facets of  $P$ , its faces of codimension 1, each of which is regarded as a polytope in its own right.

```
> Facets(P);
[
  2-dimensional polytope with 3 vertices:
    ( 1, -3,  5),
    (-2,  2, -5),
    ( 0,  1,  0),
  2-dimensional polytope with 3 vertices:
    ( 1, -3,  5),
    (-2,  2, -5),
    ( 1,  0,  0),
  2-dimensional polytope with 3 vertices:
    (1, -3,  5),
    (0,  1,  0),
    (1,  0,  0),
  2-dimensional polytope with 3 vertices:
    (-2,  2, -5),
```

```

      ( 0,  1,  0),
      ( 1,  0,  0)
]

```

Computing the (integral) points contained in a polytope (including points on its boundary) is one of the principal operations for polytopes.

```

> Points(P);
[
  (-2, 2, -5),
  (0, 0, 0),
  (0, 1, 0),
  (1, -3, 5),
  (1, 0, 0)
]

```

The interior points or boundary points can be retrieved separately using `InteriorPoints(P)` and `BoundaryPoint(P)`.

One can also compute the volume of a polytope.

```

> Volume(P);
20

```

The volume is the lattice-normalised volume:  $\text{Vol}(P) = n! \times \text{vol}(P)$ , where  $\text{vol}(P)$  is the Euclidean volume of  $P$ .

The polar, or dual, to  $P$  can be computed.

```

> D := Polar(P);
> D;
3-dimensional polytope D with 4 vertices:
  ( 3,  -1, -7/5),
  (-1,   3,  9/5),
  (-1,  -1,  1/5),
  (-1,  -1, -3/5)

```

The polar of  $P$  is again a polytope in this case (simply because the origin lies in the strict interior of  $P$ ), although its vertices need not be integral.

The Ehrhart series,

$$\text{Ehr}(D) = \sum_{n \geq 0} \#(nD \cap \mathbf{Z}^n) t^n,$$

computes the number of points in multiples of a polytope.

```

> EhrhartCoefficients(D,10);
[ 1, 7, 33, 91, 193, 355, 585, 899, 1309, 1827, 2469 ]

```

The (infinite) Ehrhart series can be recovered as a rational function.

```

> E<t> := EhrhartSeries(D);
> E;
(t^7 + 4*t^6 + 15*t^5 + 12*t^4 + 12*t^3 + 15*t^2 + 4*t + 1)/(t^8 - 3*t^7 + 3*t^6
- t^5 - t^3 + 3*t^2 - 3*t + 1)

```

It is possible to make maps of the underlying lattices and apply them to polyhedra.

```

> L := Ambient(P);

```

```

> K := Quotient(L![1,2,3]);
> K;
2-dimensional toric lattice K = (Z^3) / <1, 2, 3>
> f := LatticeMap(L, K, Matrix(3,2,[1,0,0,1,0,0]));
> Q := Image(f,P);
> Vertices(Q);
[
  (1, 0),
  (0, 1),
  (1, -3),
  (-2, 2)
]

```

In this case the image polytope  $Q = f(P)$  is not a simplex. We can triangulate it.

```

> Triangulation(Q);
{
  2-dimensional polytope with 3 vertices:
    (1, 0),
    (1, -3),
    (0, 1)
  2-dimensional polytope with 3 vertices:
    (1, -3),
    (-2, 2),
    (0, 1)
}

```

---

Our second example illustrates the machinery available for cones and polyhedra more generally.

---

#### Example H143E2

---

A ray is a rational half-line  $\mathbf{Q}^+v$  from the origin passing through a point  $v \in \mathbf{Q}^n$ . A cone is the convex hull of finitely many rays.

```

> C := Cone([[2,2],[1,1/2]]);
> C;
Cone C with 2 generators:
  ( 2,  2),
  ( 1, 1/2)

```

Although any non-zero point on a ray is enough to define it, often the non-zero integral point nearest the origin is preferred. As with polytopes, there is no analysis of a cone  $C$  on its construction, but the minimal generators are displayed once some function has forced their calculation.

```

> Rays(C);
[
  (1, 1),
  (2, 1)
]

```

```
> C;
2-dimensional simplicial cone C with 2 minimal generators:
  (1, 1),
  (2, 1)
```

A polyhedron is the Minkowski sum (simply denoted by a plus sign) of a polytope and a cone.

```
> P := StandardSimplex(2);
> P + C;
2-dimensional polyhedron with 2-dimensional finite part with 3
vertices:
  (1, 0),
  (0, 1),
  (0, 0)
and 2-dimensional infinite part given by a cone with 2 minimal
generators:
  (2, 1),
  (1, 1)
```

The polyhedron  $P + C$  is now subject to standard combinatorial analysis, such as computing its facets; some of these may be (compact) polytopes, while others may themselves be polyhedra described as the sum of a compact, or finite, part and a cone, or infinite part.

```
> Facets(P + C);
[
  1-dimensional polytope with 2 vertices:
    (1, 0),
    (0, 0),
  1-dimensional polyhedron with 0-dimensional finite part with one
  vertex:
    (0, 1)
  and 1-dimensional infinite part given by a cone with one minimal
  generator:
    (1, 1),
  1-dimensional polytope with 2 vertices:
    (0, 1),
    (0, 0),
  1-dimensional polyhedron with 0-dimensional finite part with one
  vertex:
    (1, 0)
  and 1-dimensional infinite part given by a cone with one minimal
  generator:
    (2, 1)
]
```

---

## 143.2 Polytopes, Cones and Polyhedra

A (*rational*) *polytope* (in a lattice  $L = \mathbf{Q}^n$ ) is the convex hull of finitely many points in  $L$ . (The points do not need to be integral points.)

A *cone*  $C$  (in a lattice  $L$ ) is the convex hull of finitely many rays (or zero). Precisely, a *ray* is a half line emanating from the origin, but, as is common, we treat rays synonymously with the first integral lattice point on the ray outside the origin—thus, for example, an intrinsic that returns a ray will actually return an primitive integral point of the ambient lattice. Rays that are the intersection of a linear hyperplane with  $C$  are called *extreme rays* of  $C$ —these are the corner edges of  $C$ . A cone is *regular* if it is generated (as a semigroup in the  $\mathbf{Z}$ -module  $\mathbf{Z}^n \subset L$ ) by its extreme rays (in other words, by the integral vectors on its extreme rays).

Combining all concepts so far, a (*rational*) *polyhedron* is the Minkowski sum of a polytope and a cone—the latter is the *tail cone* and is unique; the smallest possible polytope is unique but seems not to be used in the theory. We also use the expressions *compact part* and *infinite part* to indicate the polytope and the cone used to define a polyhedron.

### 143.2.1 Polytopes

Polytope( $Q$ )

The polytope defined by taking the convex hull of the sequence of points  $Q$ , where the points can be specified as sequences of integers or rational numbers (of some fixed length) or as points of a common lattice.

RandomPolytope( $L, n, k$ )

RandomPolytope( $d, n, k$ )

A polytope in the toric lattice  $L$  (or a toric lattice of dimension  $d$ ) generated by  $n$  random integral points with coefficients of modulus at most the non-negative integer  $k$ .

Polar( $P$ )

The polar dual polyhedron to the polyhedron  $P$ , namely

$$P^* = \{u \in L^\vee \mid u \cdot v \geq -1 \ \forall v \in P\}.$$

CrossPolytope( $L$ )

CrossPolytope( $d$ )

The maximum dimensional cross-polytope in the lattice  $L$ , or of dimension  $d$  for an integer  $d$ .

StandardSimplex( $L$ )

StandardSimplex( $d$ )

The simplex given by the standard basis and the origin of the lattice  $L$ , or the  $d$ -dimensional simplex given by the standard basis and the origin for an integer  $d$ .

<code>CyclicPolytope(L,n)</code>
----------------------------------

<code>CyclicPolytope(d,n)</code>
----------------------------------

The cyclic polytope generated by  $n$  points in the lattice  $L$ , or by  $n$  points in  $d$ -dimensional space for an integer  $d$ .

### 143.2.2 Cones

Cones in lattices are of type `TorCon`.

<code>Cone(A)</code>
----------------------

The cone in a lattice generated by a sequence  $A$  of its elements (or simply of sequences of integer or rational coefficients).

<code>Cone(v)</code>
----------------------

The cone in a lattice generated by a single element  $v$  of that lattice; namely the single ray  $\mathbf{Q}^+v$ .

<code>ConeWithInequalities(B)</code>
--------------------------------------

The cone in a lattice  $L$  defined by the set  $B$  of elements of the dual lattice  $L^\vee$  (or simply by a set of sequences of integer or rational coefficients). The cone is the intersection of half-spaces  $v \cdot u \geq 0$  as  $v$  runs through  $B$ :

$$\bigcap_{v \in B} \{u \in L \mid v \cdot u \geq 0\}.$$

<code>FullCone(L)</code>
--------------------------

<code>FullCone(n)</code>
--------------------------

The cone which is the entire lattice  $L$ , or the lattice of dimension  $n$  for a positive integer  $n$ .

<code>PositiveQuadrant(L)</code>
----------------------------------

<code>PositiveQuadrant(n)</code>
----------------------------------

The cone in the lattice  $L$  generated by the standard basis vectors of  $L$ , or that of the lattice of dimension  $n$  for a positive integer  $n$ .

<code>ZeroCone(L)</code>
--------------------------

<code>ZeroCone(n)</code>
--------------------------

The cone which consists of only the origin of the lattice  $L$ , or of the lattice of dimension  $n$  for a positive integer  $n$ .

Dual(C)

The dual cone to the cone  $C$ , namely

$$C^\vee = \{u \in L^\vee \mid v \cdot u \geq 0 \text{ for all } v \in C\}.$$

where  $C$  lies in the lattice  $L$ .

NormalisedCone(P)

A cone  $C$  such that the polyhedron  $P$  is the intersection of  $C$  with a hyperplane at height one, together with the embedding of the ambient lattice of  $P$  into the ambient lattice of  $C$ .

ConeInSublattice(C)

The cone of maximal dimension given by the intersection of the cone  $C$  with its linear span. Also gives the embedding of the sublattice in the ambient toric lattice.

ConeQuotientByLinearSubspace(C)

The strictly convex cone given by the quotient of the cone  $C$  by its maximal linear subspace. Also gives the quotient map.

### 143.2.3 Polyhedra

Polytopes and polyhedra in lattices are of type `TorPol`.

Polyhedron(C,H,h)

Polyhedron(C,H,h)

The polyhedron constructed as the slice of the cone  $C$  by the hyperplane determined by the primitive dual vector  $H$  at height given by the integer or rational number  $h$ .

Polyhedron(C)

**level**

`RNGINTELT`

*Default : 1*

The polyhedron arising as the intersection of the cone  $C$  with the hyperplane at height one (can be changed via ‘level’).

HalfspaceToPolyhedron(v,h)

HalfspaceToPolyhedron(Q,h)

The halfspace  $\{u \mid v \cdot u \geq h\}$  as a polyhedron, where  $v$  is a point of a toric lattice (or a sequence of integral or rational numbers that are its coefficients) and  $h \in \mathbf{Q}$ .

HyperplaneToPolyhedron(v,h)

HyperplaneToPolyhedron(Q,h)

The hyperplane  $\{u \mid v \cdot u = h\}$  as a polyhedron, where  $v$  is a point of a toric lattice (or a sequence of integral or rational numbers that are its coefficients) and  $h \in \mathbf{Q}$ .



**Polyhedron(C,f,v)**

The polyhedron arising as the preimage of the cone  $C$  under the affine map  $f + v$ .

**EmptyPolyhedron(L)**

The empty polyhedron in the toric lattice  $L$ .

**ConeToPolyhedron(C)**

The cone  $C$  as a polyhedron.

**PolyhedronInSublattice(P)**

The polyhedron of maximal dimension given by the intersection of the toric lattice containing the polyhedron  $P$  with an affine sublattice of dimension equal to the dimension of  $P$ . Also gives the affine embedding as a lattice embedding and translation.

**FixedSubspaceToPolyhedron(G)**

**FixedSubspaceToPolyhedron(L,G)**

The subspace (realised as a polyhedron) fixed by the action of the matrix group  $G$  on the toric lattice  $L$ .  $G$  should be a subgroup of either  $GL(n, \mathbf{Z})$  or of  $SL(n, \mathbf{Z})$ , where  $n$  is the dimension of  $L$ .

### Example H143E3

---

We construct some polyhedra by taking slices of the positive quadrant in 3-space.

```
> C := PositiveQuadrant(3);
> C;
3-dimensional simplicial cone C with 3 minimal generators:
(1, 0, 0),
(0, 1, 0),
(0, 0, 1)
```

We slice using dual vectors, so we recover the dual lattice  $M$  to the ambient lattice of  $C$ . Some slices are compact, some are not.

```
> M := Dual(Ambient(C));
> P := Polyhedron(C, M ! [1,2,3], 1);
> IsPolytope(P);
true
> Q := Polyhedron(C, M ! [1,-2,3], 1);
> IsPolytope(Q);
false
> Q;
2-dimensional polyhedron Q with 1-dimensional finite part with 2 vertices:
( 1, -1),
( 0, -1/3)
and 2-dimensional infinite part given by a cone with 2 minimal generators:
```

```
( 2,  -1),
( 0,   1)
```

We can change the level, or height, at which the slice is taken.

```
> R := Polyhedron(C, M ! [1,2,3], -5);
> R;
Empty polyhedron
```

---

#### 143.2.4 Arithmetic Operations on Polyhedra

**C eq D**

Return **true** if and only if the cones  $C$  and  $D$  are equal, that is, they lie in the same toric lattice and have identical supports.

**P eq Q**

Return **true** if and only if the polyhedra  $P$  and  $Q$  are equal, that is, they lie in the same toric lattice and have identical supports.

**C meet D**

The intersection of the cones  $C$  and  $D$ ; the cones must lie in the same toric lattice otherwise an error is reported.

**P meet Q**

The intersection of the polyhedra  $P$  and  $Q$ ; the polyhedra must lie in the same toric lattice otherwise an error is reported.

**P subset Q**

Return **true** if and only if the support of the polyhedron  $P$  is contained in that of the polyhedron  $Q$ ; the polyhedra must lie in the same toric lattice otherwise an error is reported.

**C + D**

**P + Q**

The Minkowski sum of the cones  $C$  and  $D$  or of the polyhedra  $P$  and  $Q$ .

**P + C**

**C + P**

The polyhedron constructed as the Minkowski sum of the polytope  $P$  and the cone  $C$ .

**P \* Q**

The convex hull of the Cartesian product of the polyhedra  $P$  and  $Q$ .

$k * P$

The dilation of the polyhedron  $P$  by the rational number  $k$ .

$-P$

The negation of the polyhedron  $P$ .

### 143.3 Basic Combinatorics of Polytopes and Polyhedra

Cones, polytopes and polyhedra have similar combinatorial features: they are composed of faces of various dimensions that meet in faces of lower dimensions. Some functions apply to all of these geometrical objects, some do not.

A face of a cone  $C$  is the intersection of  $C$  with an affine hyperplane whose defining equation is non-negative on  $C$ . A facet of a cone  $C$  is a codimension 1 face of  $C$ . Definitions for polytopes and polyhedra are similar.

#### 143.3.1 Vertices and Inequalities

`Vertices(P)`

A sequence containing the vertices of the polyhedron  $P$ .

`NumberOfVertices(P)`

The number of vertices of the polyhedron  $P$ .

`Rays(C)`

The sequence of generators of the rays of the cone  $C$  (returned as primitive lattice points). If  $C$  is strictly convex, this is the same as the minimal  $\mathbf{R}$ -generators.

`Ray(C,i)`

The  $i$ th ray of the of the cone  $C$  (in the order returned by `Rays(C)`).

`LinearSpanEquations(C)`

`LinearSpanEquations(Q)`

A sequence of equations defining the minimal linear subspace containing the cone  $C$  (or the sequence of toric lattice points  $Q$ ).

`LinearSpanGenerators(C)`

`LinearSpanGenerators(Q)`

A sequence of generators of the minimal linear subspace containing the cone  $C$  (or the sequence of toric lattice points  $Q$ ).

`LinearSubspaceGenerators(C)`

A basis of the maximal linear subspace contained in the cone  $C$ .

Inequalities(C)
-----------------

Inequalities(P)
-----------------

MinimalInequalities(C)
------------------------

If the cone  $C$  or polyhedron  $P$  lies in a toric lattice  $L$ , then return a finite sequence of vectors in the dual lattice  $\check{L}$  which define supporting hyperplanes of  $C$  or  $P$ . The list is forced to be minimal for `MinimalInequalities`, but might not be otherwise. If the argument is a polyhedron, then an integer  $k$  is returned as a second return value, so that the first  $k$  inequalities correspond to the facets of  $P$  while the remaining cut out the subspace containing  $P$ .

---

**Example H143E4**

Build a polytope and recover the inequalities that define it: the second return value, 3, says that all three inequalities define facets of  $P$ —we see an example below where this is not the case.

```
> P := Polytope([[1,0],[0,1],[-1,-1]]);
> Inequalities(P);
[
  <(2, -1), -1>,
  <(-1, -1), -1>,
  <(-1, 2), -1>
]
3
```

For each inequality use `HalfspaceToPolyhedron(m,h)` to define the corresponding halfspace, and then intersect them all to recover  $P$ —and finally recover the vertices we started with.

```
> PP := &meet [HalfspaceToPolyhedron(H[1],H[2]) : H in Inequalities(P)];
> PP eq P;
true
> PP;
2-dimensional polytope PP with 3 vertices:
( 0,  1),
(-1, -1),
( 1,  0)
```

It can happen that a polytope does not span the ambient toric lattice in which it lies and that some of the inequalities are used to cut out the affine subspace that it does span.

```
> Q := Polytope([[1,0,2],[0,1,2],[-1,0,2],[0,-1,2]]);
> Inequalities(Q);
[
  <(1, -1, 0), -1>,
  <(-1, -1, 0), -1>,
  <(-1, 1, 0), -1>,
  <(1, 1, 0), -1>,
  <(0, 0, 1), 2>,
  <(0, 0, -1), -2>
```

]
   
4

In this case the final two inequalities are opposites of one another and cut out the affine hyperplane  $z = 2$ . The second return value 4 indicates that the first 4 inequalities are cutting out facets of the polytope while the remaining inequalities are cutting out the affine hyperplane.

---

### 143.3.2 Facets and Faces

`#fVector(C)`

`fVector(P)`

The  $f$ -vector of the polyhedron  $P$  or cone  $C$ .

`#hVector(C)`

`#hVector(P)`

The  $h$ -vector of the polyhedron  $P$  or cone  $C$ .

`Facets(C)`

`Facets(P)`

A sequence containing all facets of the toric cone  $C$  or polyhedron  $P$ .

`FacetIndices(P)`

A sequence of sets describing the facets of the polytope  $P$ . The  $j$ th set gives the indices of the vertices of  $P$  which define the  $j$ th facet of  $P$ .

`NumberOfFacets(P)`

The number of facets of the polyhedron  $P$ .

`Faces(C)`

`Faces(P)`

`Faces(C,i)`

`Faces(P,i)`

A sequence containing all face cones of the toric cone  $C$  or polyhedron  $P$ , or only those of dimension  $i$  if an integer  $i$  is also specified.

`FaceIndices(P,i)`

A sequence of sets describing the  $i$ -dimensional faces of the polyhedron  $P$ . The  $j$ th set gives the indices of the vertices of  $P$  which define the  $j$ th  $i$ -dimensional face.

`Edges(P)`

A sequence containing all the edges of the polyhedron  $P$ .

**EdgeIndices(P)**

A sequence of sets describing the edges of the polyhedron  $P$ . The  $j$ th set gives the indices of the vertices of  $P$  which define the  $j$ th edge of  $P$ .

**Graph(P)**

The graph of the face lattice of the polyhedron  $P$ . The vertices of the graph are labeled by the dimension of the corresponding face.

**FaceSupportedBy(C,H)**

The face of the toric cone  $C$  supported by the toric lattice element  $H$  in the dual lattice to the one containing  $C$  (so  $H$  is a linear form on the ambient lattice of  $C$ ).

**IsSupportingHyperplane(v,h,P)**

Return **true** if and only if the hyperplane defined by  $v \cdot u = h$  is a supporting hyperplane of the polyhedron  $P$ , where  $v$  is a lattice point of the dual ambient lattice of  $P$  and  $h$  is a rational number. If so, also gives the sign  $\tau$  such the hyperplane is a support of  $P$  (i.e.  $\tau$  in  $\{-1, 0, +1\}$  such that  $\text{Sign}(v \cdot u - h)$  is either 0 or  $\tau$  for all  $u$  in  $P$ ). If  $P$  is contained within the hyperplane, then  $\tau$  will be 0.

**SupportingCone(P,v)**

The cone  $C$  such that  $C + v$  is a supporting cone of the polyhedron  $P$ , where  $v$  is a vertex of  $P$ .

## 143.4 The Combinatorics of Polytopes

### 143.4.1 Points in Polytopes

Given a polygon, one might want to know how many integral points it contains, and one might want to list them. These are computed by different algorithms (although one can of course list the points and then count them). For point counting we use the methods of Barvinok and Pommersheim ([BP99]), as described in [DLHTY04]. More generally, one might want to know the number of points in integral dilations of a polytope: these numbers are the coefficients of a generating function, the Ehrhart series; this is discussed in a later section.

**Points(P)****InteriorPoints(P)****BoundaryPoints(P)**

The integral toric lattice points, the strictly interior lattice points, or the boundary lattice points of the polytope  $P$ .

**NumberOfPoints(P)**

The number of integral toric lattice points of the polytope  $P$ .

### 143.4.2 Ehrhart Theory of Polytopes

**EhrhartSeries(P)**

The rational generating function of the Ehrhart series for the polytope  $P$ .

**EhrhartPolynomial(P)**

A sequence representing the Ehrhart (quasi-)polynomial for the polytope  $P$ . That is, a sequence of polynomials  $[p_0, \dots, p_{r-1}]$  of length  $r$ , the quasi-period of the Ehrhart polynomial, so that the number of lattice points of the dilation  $nP$  is the value of  $p_s(k)$  where  $n = kr + s$  is the Euclidean division of  $n$  by  $r$ ; in other words,  $s$  is the least residue of  $n$  modulo  $r$ . Note that since MAGMA indexes sequences from 1, we have that  $p_i = \text{EhrhartPolynomial}(P)[i+1]$ .

**EhrhartCoefficients(P,l)**

The first  $l + 1$  coefficients of the Ehrhart series for the polytope  $P$  (starting with  $0P$  up to and including  $lP$ ).

**EhrhartCoefficient(P,k)**

The number of lattice points in the (non-negative, integral) dilation  $kP$  of the polytope  $P$ .

### 143.4.3 Automorphisms of a Polytope

**AutomorphismGroup(P)**

The subgroup of  $GL(n, \mathbf{Z})$  (acting on the ambient lattice) which leaves the polyhedron  $P$  unchanged.

---

#### Example H143E5

```
> P:=CrossPolytope(2);
> P;
2-dimensional polytope P with 4 vertices:
  ( 1,  0),
  ( 0,  1),
  (-1,  0),
  ( 0, -1)
> AutomorphismGroup(P);
MatrixGroup(2, Integer Ring)
Generators:
  [0 1]
  [1 0]
  [ 0  1]
  [-1  0]
```

---

### 143.4.4 Operations on Polytopes

**Triangulation(P)**

A sequence of polytopes that make a triangulation of the polytope  $P$ .

**TriangulationOfBoundary(P)**

A sequence of polytopes that make a triangulation of the boundary of the polytope  $P$ .

## 143.5 Cones and Polyhedra

### 143.5.1 Generators of Cones

**ZGenerators(C)**

**level**

RNGINTELT

*Default : infinity*

If  $C$  lies in a toric lattice  $L$ , then return a finite sequence of elements of  $L$  that generate  $C$  as a monoid. If specified, the parameter **level** restricts the search for generators to that given level; this assumes that  $C$  is graded.

**RGenerators(C)**

**MinimalRGenerators(C)**

If  $C$  lies in a toric lattice  $L$ , then return a finite sequence of elements of  $L$  that generate  $C$  over  $\mathbf{Q}_+$ . The list is forced to be minimal for **MinimalRGenerators**, but might not be otherwise.

**Points(C,H,h)**

The set of integral lattice points in the cone  $C$  contained in the hyperplane given by the section determined by the dual lattice element  $H$  at height given by the rational number  $h$ . The intersection of  $H$  with  $C$  is required to be compact.

#### Example H143E6

---

Here we find the generators of a cone in the sublattice spanned by its defining lattice points. First build some points in a toric lattice and make the cone they generate.

```
> L := ToricLattice(4);
> B := [ L | [1,2,3,-3], [-1,0,1,0], [1,2,1,-2], [2,0,0,-1], [0,0,2,-1] ];
> L1,f := Sublattice(B);
> Dimension(L1);
3
> C := Cone(B);
> Points(Polytope(B));
{
  (1, 2, 1, -2),
  (2, 0, 0, -1),
```



```

      (-1, 0, 1, 0),
      (1, 2, 3, -3),
      (1, 0, 1, -1),
      (0, 0, 2, -1),
      (0, 1, 1, -1),
      (1, 1, 2, -2)
    }
  > ZGenerators(C);
  [
    (-1, 0, 1, 0),
    (0, 1, 1, -1),
    (1, 2, 1, -2),
    (2, 0, 0, -1)
  ]

```

Using the embedding map  $f$ , pull the cone back to the lattice span of  $C$ .

```
> C1 := C @@ f;
```

In the smaller lattice, this cone is nonsingular.

```
> IsNonsingular(C1);
true
```

We can find out which points generate the cone in the smaller lattice and compare them with the generators of the original cone.

```

> B1 := ZGenerators(C1);
> B1;
[
  (-1, 0, 1),
  (1, 1, 0),
  (2, 0, -1)
]
> [ Index(B, Image(f, b)) : b in B1 ];
[ 2, 3, 4 ]

```

### 143.5.2 Properties of Polyhedra

#### CompactPart(P)

The polytope defined by taking the convex hull of the vertices of the polyhedron  $P$ . This is the smallest polytope which can occur as a factor in an expression of  $P$  as the Minkowski sum of a polytope and a cone.

#### IntegralPart(P)

The polyhedron defined by taking the convex hull of the integral points contained in the polyhedron  $P$ .

InfinitePart( $P$ )

The tail cone of the polyhedron  $P$ .

IsEmpty( $P$ )

Return **true** if and only if the polyhedron  $P$  is empty.

---

**Example H143E7**

One can build a polytope as the convex hull of a finite sequence of points—these may be elements of a toric lattice or simply sequences of integer or rational coefficients.

```
> P := Polytope([[ -4, 2, 1], [0, 3, 4], [3, 1, -3], [3, 0, 0], [2, -1, 1]]);
```

The polar polyhedron comprises dual vectors that evaluate to at least  $-1$  on  $P$ . It is compact if and only if the origin lies in the interior of  $P$ .

```
> D := Polar(P);
```

```
> D;
```

3-dimensional polyhedron  $D$  with 3-dimensional finite part with 5 vertices:

```
( 1/2,    1,    -1),
(-1/3,   1/2,   1/6),
(-1/3,   1/21, -2/7),
(-1/3,  -3/13, -1/13),
(1/67, -37/67, 11/67)
```

and 3-dimensional infinite part given by a cone with 3 minimal generators:

```
(  1,    2,    0),
(  2,    9,    5),
(  7,    9,   10)
```

Since  $D$  is infinite, there is no intrinsic to list its integral points, but one can determine whether it contains integral points at all.

```
> HasIntegralPoint(D);
```

```
true
```

Computing the span of all the integral points of  $D$  is possible.

```
> IntegralPart(D);
```

3-dimensional polyhedron with 3-dimensional finite part with 6 vertices:

```
(0, 0, 0),
(0, 1, 0),
(0, 2, 1),
(1, 1, 1),
(1, 2, -1),
(2, 2, 3)
```

and 3-dimensional infinite part given by a cone with 3 minimal generators:

```
(1, 2, 0),
(2, 9, 5),
(7, 9, 10)
```

The compact and infinite parts of  $D$  can be recovered.

```
> cD := CompactPart(D);
```

```

> cD;
3-dimensional polytope cD with 5 vertices:
  ( 1/2,    1,    -1),
  (-1/3,   1/2,   1/6),
  (-1/3,  1/21, -2/7),
  (-1/3, -3/13, -1/13),
  (1/67, -37/67, 11/67)

```

Polytopes are special cases of polyhedra (those whose infinite tail cone is the zero cone, or equivalently those that are compact), and the distinction can be determined.

```

> IsPolytope(cD);
true
> IsPolytope(D);
false

```

### Example H143E8

---

```

> C:=Cone([[0,1,0],[0,1,1],[1,1,2],[1,1,4]]);
> P:=Polyhedron(C);
> P;
2-dimensional polyhedron P with 1-dimensional finite part with 2
vertices:
  (1, 2),
  (1, 4)
and 2-dimensional infinite part given by a cone with 2 minimal
generators:
  (1, 0),
  (1, 1)

> CC:=Cone([[1,0],[1,1]]);
> QQ:=Polytope([[1,2],[1,4]]);
> PP:=CC + QQ;
> PP;
2-dimensional polyhedron PP with 1-dimensional finite part with 2
vertices:
  (1, 2),
  (1, 4)
and 2-dimensional infinite part given by a cone with 2 minimal
generators:
  (1, 0),
  (1, 1)

```

But there's a potential catch.

```

> PP eq P;
false
> Ambient(PP);
2-dimensional toric lattice Z^2

```

```

> Ambient(P);
2-dimensional toric lattice ker <1, 0, 0>
> P:=ChangeAmbient(P,Ambient(PP));
> PP eq P;
true

```

The ambients are different simply because of the method of construction of the two polyhedra, but they can be forced into the same space.

---

IsMaximalDimension(C)
-----------------------

IsMaximalDimension(P)
-----------------------

Return **true** if and only if the cone  $C$  or polyhedron  $P$  has dimension equal to that of its ambient lattice.

IsStrictlyConvex(C)
---------------------

Return **true** if and only if the cone  $C$  is strictly convex; that is, if there exists a hyperplane  $H$  such that  $C$  is contained on one side of  $H$  and  $C$  meets  $H$  in single point 0.

IsSimplicial(C)
-----------------

IsSimplicial(P)
-----------------

Return **true** if and only if the cone  $C$  or the polyhedron  $P$  is simplicial.

IsSimplex(P)
--------------

Return **true** if and only if the polyhedron  $P$  is a simplex.

IsSimple(P)
-------------

Return **true** if and only if the polyhedron  $P$  is simple.

IsAffineLinear(P)
-------------------

Return **true** if and only if the polyhedron  $P$  is an affine linear space.

IsZero(C)
-----------

Return **true** if and only if the cone  $C$  is supported at the origin of its ambient toric lattice.

### 143.5.3 Attributes of Polyhedra

Dimension(C)
--------------

Dimension(P)
--------------

The dimension of the toric cone  $C$  or polyhedron  $P$ .

Index(C)
----------

The index of the sublattice generated by the minimal  $\mathbf{R}$ -generators of the cone  $C$  in its linear span.

IsShellable(P)
----------------

Return **true** if and only if the integral polytope  $P$  is shellable; i.e. if there exists a polytope  $S$  such that each lattice point  $u \in P$  lies on the boundary of  $v + i * (S - v)$  for some  $0 \leq i \leq k$ , where  $v$  is the barycentre of the vertices of  $P$ . If **true**, also returns  $S$ ,  $v$ , and  $k$ .

### 143.5.4 Combinatorics of Polyhedral Complexes

Ambient(C)
------------

Ambient(P)
------------

The ambient toric lattice of the toric cone  $C$  or polyhedron  $P$ .

ChangeAmbient(C,L)
--------------------

ChangeAmbient(P,L)
--------------------

Make a cone or polyhedron identical to the toric cone  $C$  or polyhedron  $P$  but lying in the toric lattice  $L$ ; the identification of the existing ambient toric lattice with  $L$  is simply by identification of the two standard bases and it requires the dimensions of the two spaces to be equal.

### 143.6 Toric Lattices

One often begins to study toric geometry by considering a ‘lattice’  $L = \mathbf{Z}^n$  and discussing polygons or cones in its overlying rational (or real) vector space  $L_{\mathbf{Q}} = L \otimes \mathbf{Q}$  whose vertices lie on  $L$ ; these are examples of so-called ‘lattice polytopes’. Anybody who has given a seminar on toric geometry will know the constant frustration of distinguishing between  $L$  and  $L_{\mathbf{Q}}$  (or worse, a bit later in the story, the duals of these). In MAGMA, we bind these two spaces together in a single object, a *toric lattice*. These spaces lie underneath all the combinatorics. It is not often necessary to be explicit about them, since the package handles them invisibly in the background, but they are useful for the usual parent and data typing purposes: when one creates an empty sequence intended to house elements of a toric lattice, it should be properly typed from the outset, for instance. Toric lattices created as duals record that relationship.

Toric lattices are of type **TorLat** and their elements are of type **TorLatElt**.

### 143.6.1 Toric Lattices

A toric lattice is a finite-dimensional rational vector space with a distinguished free  $\mathbf{Z}$ -module  $L$  that spans it: it is the pair  $L \otimes \mathbf{Q} \supset L$  where  $L \cong \mathbf{Z}^n$ . We usually refer to the toric lattice as  $L$ , and although the feeling of using  $L$  is that it is simply the given  $\mathbf{Z}$ -module, the covering vector space allows fluent working with non-integral points of  $L$ .

ToricLattice(n)

Create an  $n$ -dimensional toric lattice  $\mathbf{Q}^n \supset \mathbf{Z}^n$ . The integer  $n$  must be non-negative.

ScalarLattice()

The unique one-dimensional toric lattice of scalars  $\mathbf{Q} \supset \mathbf{Z}$ .

---

#### Example H143E9

A properly typed sequence of points of a toric lattice can be summed with the usual convention.

```
> L := ToricLattice(3);
> nopoints := [ L | ];
> &+ nopoints;
(0, 0, 0)
```

A toric lattice is really a vector space marked with a finitely-generated  $\mathbf{Z}$ -module that spans it. In particular, its points may well have rational coefficients, although those with integral coefficients are distinguished.

```
> somepoints := [ L | [1/2, 2/3, 3/4], [1, 2, 3] ];
> [ IsIntegral(v) : v in somepoints ];
[ false, true ]
```

---

Dual(L)

The dual lattice of the toric lattice  $L$ .

---

#### Example H143E10

The dual of a toric lattice is also a toric lattice: integrality of dual points is clear.

```
> L := ToricLattice(2);
> L;
2-dimensional toric lattice L = Z^2
> M := Dual(L);
> M;
2-dimensional toric lattice M = (Z^2)^*
```

MAGMA's default printing for toric lattices tries to help keep track of which lattice is which—this becomes more useful when working with toric varieties. The relationship between  $L$  and  $M$  is preserved, and double-dual returns  $L$ .

```
> M eq L;
false
```

```
> L eq Dual(M);
true
```

---

$L + M$
---------

<code>DirectSum(L,M)</code>
-----------------------------

<code>DirectSum(Q)</code>
---------------------------

The direct sum of the two toric lattices  $L$  and  $M$ . The following natural maps are also returned: the embedding of  $L$  into the sum, the embedding of  $M$ , the projection of the sum onto  $L$ , and the projection onto  $M$ .

The argument can also be a sequence  $Q$  of toric lattices. In this case, there are three return values: the direct sum of all lattices in  $Q$ , a sequence of inclusion maps of lattices in  $Q$  into the sum, and a sequence of projection maps from the sum onto each of the elements of  $Q$  in turn.

$L \wedge n$
--------------

The direct sum of the toric lattice  $L$  with itself  $n$  times. Also return are a sequence of injections of  $L$  as factors into the sum and a sequence of projections of the sum onto its factors. This is identical to `DirectSum([L,L,...,L])` where the sequence has length  $n$ .

<code>Dimension(L)</code>
---------------------------

The dimension of the toric lattice  $L$ .

### 143.6.2 Points of Toric Lattices

Points of  $L$  are interpreted to mean points of  $L \otimes \mathbf{Q}$ . If the coefficients of a point are in fact integral, then this is recognised: so it is possible to tell whether a point is in fact a point of  $L$ , not just the rational span.

The usual rational vector space arithmetic operates on these lattice points.

$L \text{ ! } [a,b,\dots]$
----------------------------

<code>LatticeVector(L,Q)</code>
---------------------------------

<code>LatticeVector(Q)</code>
-------------------------------

The point  $(a,b,\dots)$  as an element of the toric lattice  $L$ , where  $Q = [a,b,\dots]$  is a sequence of rational numbers or integers. (If the toric lattice  $L$  is omitted, then it will be created.)

$L \cdot i$
-------------

<code>Basis(L,i)</code>
-------------------------

The  $i$ th standard basis element of the toric lattice  $L$ .

Basis(L)
----------

The standard basis element of the toric lattice  $L$  as a sequence of points of  $L$ .

Form(L,Q)
-----------

The point  $(a, b, \dots)$  as an element of the dual of the toric lattice  $L$ , where  $Q = [a, b, \dots]$  is a sequence of rational numbers or integers.

Zero(L)
---------

The zero vector in the toric lattice  $L$ .

$P + Q$
---------

$P - Q$
---------

$n * P$
---------

$P / n$
---------

The sum and difference of toric lattice points  $P$  and  $Q$  (or points of the dual), the rational multiple  $nP$  and quotient  $P/n$ , where  $n \in \mathbf{Q}$ .

$P \text{ eq } Q$
-------------------

Return **true** if and only if the toric lattice points  $P$  and  $Q$  are the same point of the same lattice.

AreProportional(P,Q)
----------------------

Return **true** if and only if the toric lattice points  $P$  and  $Q$  are rational multiples of one another; the factor  $Q/P$  is returned in that case.

$P / Q$
---------

The rational factor  $P/Q$  in the case that the toric lattice points  $P$  and  $Q$  are rational multiples of one another.

### Example H143E11

---

This simple example builds some points in a toric lattice and performs arithmetic on them.

```
> L := ToricLattice(3);
> a := L ! [1,2,3];
> a;
(1, 2, 3)
> L eq Parent(a);
true
> b := L ! [1/2,1,3/2];
> a + b;
(3/2, 3, 9/2)
> a eq b;
false
> a eq 2*b;
true
```



```
> b/a;
1/2
```

`v in L`

Return **true** if and only if the toric lattice point  $v$  lies in the toric lattice  $L$ .

`IsZero(v)`

Return **true** if and only if the toric lattice point  $v$  is the zero vector.

`IsIntegral(v)`

Return **true** if and only if the coefficients of the toric lattice point  $v$  are integral.

`IsPrimitive(v)`

Return **true** if and only if the toric lattice point  $v$  is a primitive integral vector; that is,  $v$  is not divisible as an integral vector in its ambient toric lattice.

`PrimitiveLatticeVector(v)`

The first toric lattice point on the ray spanned by  $v$ .

### Example H143E12

A point of a sublattice may be primitive in the sublattice even though it is not primitive in the bigger lattice: treating sublattices as the images of embeddings makes this point transparent. First build a sublattice  $K$  of  $L$  together with the embedding map.

```
> L := ToricLattice(2);
> K,emb := Sublattice([L | [2,0],[0,2]]);
```

Construct a point in  $L$  that is clearly not primitive.

```
> vL := L ! [2,2];
> IsPrimitive(vL);
false
```

Pulling this point back to  $K$  shows that it is obviously primitive in  $K$ .

```
> vK := vL @@ emb;
> vK;
(1, 1)
> IsPrimitive(vK);
true
```

But notice that coercion of the point of  $L$  into  $K$  is not the same thing: it simply works with the coefficients of the point and gives an answer that is not what is wanted in this case.

```
> K ! vL;
(2, 2)
> IsPrimitive(K ! vL);
false
```

### 143.6.3 Operations on Toric Lattices

**L eq K**

Return **true** if and only if the toric lattices  $L$  and  $K$  are the same object in MAGMA (and not merely isomorphic).

**Sublattice(Q)**

A toric lattice  $L_1$  isomorphic to the sublattice of a toric lattice  $L$  generated by the sequence  $Q$  of elements of  $L$  together with an embedding map of  $L_1$  in  $L$ . The MAGMA subobject constructor **sub**<  $L$  |  $Q$  > can also be used, although the sequence  $Q$  must be a sequence of elements of  $L$  and cannot be interpreted more broadly.

**ToricLattice(Q)**

The toric sublattice of  $\mathbf{Q}^n \supset \mathbf{Z}^n$  (regarded as a toric lattice) generated by the sequences of integers of length  $n$  of which the sequence  $Q$  comprises.

**Quotient(C)**

**Quotient(Q)**

**Quotient(v)**

A toric lattice isomorphic to the toric lattice that is the quotient of a toric lattice  $L$  by the linear span of the cone  $C$ , or the elements  $v \in L$  that comprise the sequence  $Q$  or the single primitive vector  $v \in L$ . The projection map of  $L$  to the quotient is also returned.

**AddVectorToLattice(v)**

**AddVectorToLattice(Q)**

A toric lattice  $L_1$  isomorphic to the toric lattice generated by a toric lattice  $L$  together with the vector  $v \in L$  (or by a sequence of vectors of  $L$ ). The inclusion map of  $L$  in  $L_1$  is also returned.

**IsSublattice(L)**

Return **true** if and only if the toric lattice  $L$  was constructed as a sublattice of another toric lattice.

**IsSuperlattice(L)**

Return **true** if and only if the toric lattice  $L$  was constructed as a superlattice of another toric lattice (by the addition of a rational vector).

**IsDirectSum(L)**

Return **true** if and only if the toric lattice  $L$  was constructed as the direct sum of other toric lattices.

**IsQuotient(L)**

Return **true** if and only if the toric lattice  $L$  was constructed as the quotient of another toric lattice.

**Sublattice(L)**

If the toric lattice  $L$  was constructed as the extension of a toric lattice  $K$  by the addition of a vector, then return  $K$ .

**Superlattice(L)**

If the toric lattice  $L$  was constructed as a sublattice of a toric lattice  $K$ , then return  $K$ .

**Summands(L)**

If the toric lattice  $L$  was constructed as a direct sum, return a sequence of the toric lattice summands used (and parallel sequences of their embedding maps in  $L$  and the projections from  $L$  to them).

**Example H143E13**

---

Construct a sublattice of a toric lattice  $L$  and compute its natural basis in the original coordinates of  $L$ ; the user does not have control over the choice of coordinates and inclusion map.

```
> L := ToricLattice(2);
> L1, phi1 := Sublattice([L | [1,2],[2,1]]);
> L1;
2-dimensional toric lattice L1 = sub(Z^2)
> phi1(L1.1), phi1(L1.2);
(1, 2)
(0, 3)
```

A similar calculation for a quotient map.

```
> L2, phi2 := Quotient(L ! [3/2,2]);
> L2;
1-dimensional toric lattice L2 = (Z^2) / <3, 4>
> phi2(L.1), phi2(L.2);
(-4)
(3)
```

And again for an extension of  $L$ .

```
> L3, phi3 := AddVectorToLattice(L ! [1/5,2/5]);
> L3;
2-dimensional toric lattice L3 = (Z^2) + <1/5, 2/5>
> phi3(L1.1), phi3(L1.2);
(1, 0)
(2, 5)
```

---

### 143.6.4 Maps of Toric Lattices

Maps between toric lattices are of type `TorLatMap`. They can be constructed using the `hom< L -> K | M >` constructor, where  $M$  is the matrix of the desired linear map with respect to the standard bases of  $L$  and  $K$ . The usual evaluation operations `@` and `@@` can be applied, although there are image and preimage intrinsics too. Basic arithmetic of maps (sum, difference and scalar multiple, each taken pointwise) is available.

ZeroMap(L,K)

The zero map between toric lattices  $L$  and  $K$ .

IdentityMap(L)

The identity map on the toric lattice  $L$ .

hom< L -> K | M >

LatticeMap(L,K,M)

The map between toric lattices  $L$  and  $K$  determined by the matrix  $M$  (with respect to the standard bases of  $L$  and  $M$ ).

LatticeMap(L,Q)

The toric lattice map from toric lattice  $L$  to the toric lattice  $M$  determined by the sequence  $Q$  of toric lattice points, where  $M$  is the toric lattice containing the points of  $Q$ .

DefiningMatrix(f)

The defining matrix of the lattice map  $f$ .

Image(f,C)

Image(f,P)

Image(f,v)

The image of the cone  $C$  or polyhedron  $P$  or lattice element  $v$  under the toric lattice map  $f$ ; the type of the object is preserved under the map.

Preimage(f,C)

Preimage(f,P)

Preimage(f,v)

The preimage of the cone  $C$  or polyhedron  $P$  or lattice element  $v$  by the toric lattice map  $f$ ; the type of the object is preserved.

KernelEmbedding( $f$ )
------------------------

KernelEmbedding( $v$ )
------------------------

The inclusion of the kernel of the toric lattice map  $f$ , regarded as a distinct toric lattice, into the domain of  $f$ ; or the same for the dual subspace annihilated by lattice element  $v$ .

KernelBasis( $f$ )
--------------------

KernelBasis( $v$ )
--------------------

A basis for the kernel of the toric lattice map  $f$  or the dual of the sublattice annihilated by the lattice element  $v$ .

ImageBasis( $f$ )
-------------------

A basis for the image of the toric lattice map  $f$ .

IsCokernelTorsionFree( $f$ )
------------------------------

Return **true** if and only if the cokernel of the toric lattice map  $f$  is torsion free.

## 143.7 Bibliography

- [BP99] Alexander Barvinok and James E. Pommersheim. An algorithmic theory of lattice points in polyhedra. In *New perspectives in algebraic combinatorics (Berkeley, CA, 1996–97)*, volume 38 of *Math. Sci. Res. Inst. Publ.*, pages 91–147. Cambridge Univ. Press, Cambridge, 1999.
- [DLHTY04] Jesús A. De Loera, Raymond Hemmecke, Jeremiah Tauzer, and Ruriko Yoshida. Effective lattice point counting in rational convex polytopes. *J. Symbolic Comput.*, 38(4):1273–1302, 2004.



# **PART XX**

## **COMBINATORICS**

144	ENUMERATIVE COMBINATORICS	4805
145	PARTITIONS, WORDS AND YOUNG TABLEAUX	4811
146	SYMMETRIC FUNCTIONS	4845
147	INCIDENCE STRUCTURES AND DESIGNS	4871
148	HADAMARD MATRICES	4907
149	GRAPHS	4917
150	MULTIGRAPHS	4999
151	NETWORKS	5047





# 144 ENUMERATIVE COMBINATORICS

<b>144.1 Introduction . . . . .</b>	<b>4807</b>		
<b>144.2 Combinatorial Functions . .</b>	<b>4807</b>		
Factorial(n)	4807	Bell(n)	4808
NumberOfPermutations(n, k)	4807	EulerianNumber(n, r)	4808
Binomial(n, r)	4807	HarmonicNumber(n)	4808
Multinomial(n, [r <sub>1</sub> , ... r <sub>n</sub> ])	4807	BernoulliNumber(n)	4808
Fibonacci(n)	4807	BernoulliApproximation(n)	4808
Catalan(n)	4807	BernoulliPolynomial(n)	4808
Lucas(n)	4807	<b>144.3 Subsets of a Finite Set . . .</b>	<b>4809</b>
GeneralizedFibonacci		Subsets(S)	4809
Number(g <sub>0</sub> , g <sub>1</sub> , n)	4807	Subsets(S, k)	4809
StirlingFirst(n, k)	4808	Multisets(S, k)	4809
StirlingSecond(n, k)	4808	Subsequences(S, k)	4809
		Permutations(S)	4809
		Permutations(S, k)	4809



# Chapter 144

## ENUMERATIVE COMBINATORICS

### 144.1 Introduction

This chapter presents some of the tools provided by MAGMA for enumerative combinatorics.

### 144.2 Combinatorial Functions

**Factorial(n)**

The factorial  $n!$  for non-negative small integer  $n$ .

**NumberOfPermutations(n, k)**

The number of permutations of  $n$  distinct objects taken  $k$  at a time.

**Binomial(n, r)**

The binomial coefficient  $\binom{n}{r}$ .

**Multinomial(n, [r<sub>1</sub>, ... r<sub>n</sub>])**

Given a sequence  $Q = [r_1, \dots, r_k]$  of positive integers such that  $n = r_1 + \dots + r_k$ , return the multinomial coefficient  $\binom{n}{r_1, \dots, r_k}$ .

**Fibonacci(n)**

Given an integer  $n$ , this function returns the  $n$ -th Fibonacci number  $F_n$ , which can be defined via the recursion  $F_0 = 0$ ,  $F_1 = 1$ ,  $F_n = F_{n-1} + F_{n-2}$  for all integers  $n$ . Note that  $n$  is allowed to be negative, and that  $F_{-n} = (-1)^{n+1}F_n$ .

**Catalan(n)**

Given a small non-negative integer  $n$ , this function returns the  $n$ -th Catalan number  $C_n$ , defined via the recursion  $C_0 = 1$ ,  $C_{n+1} = C_n \cdot (4n+2)/(n+2)$ .

**Lucas(n)**

Given an integer  $n$ , this function returns the  $n$ -th Lucas number  $L_n$ , which can be defined via the recursion  $L_0 = 2$ ,  $L_1 = 1$ ,  $L_n = L_{n-1} + L_{n-2}$  for all integers  $n$ . Note that  $n$  is allowed to be negative, and that  $L_{-n} = (-1)^n L_n$ .

**GeneralizedFibonacciNumber(g0, g1, n)**

The  $n$ th member of the generalized Fibonacci sequence defined by  $G_0 = g_0$ ,  $G_1 = g_1$ ,  $G_n = G_{n-1} + G_{n-2}$  for all integers  $n$ . Note that  $n$  is allowed to be negative. The Fibonacci and Lucas numbers are special cases where  $(g_0, g_1) = (0, 1)$  or  $(2, 1)$  respectively.

**StirlingFirst(n, k)**

The Stirling number of the first type,  $[n_k]$ , where  $n$  and  $k$  are non-negative integers.

**StirlingSecond(n, k)**

The Stirling number of the second type,  $\{n_k\}$ , where  $n$  and  $k$  are non-negative integers.

**Bell(n)**

The  $n$ th Bell number, giving the number of partitions of a set of size  $n$ . (Not to be confused with **NumberOfPartitions(n)**, which gives the number of partitions of the *integer*  $n$ .) This is equal to the sum of **StirlingSecond(n,k)** for  $k$  between 0 and  $n$  (inclusive).

**EulerianNumber(n, r)**

The number  $E(n, r)$  of permutations  $p$  of  $\{1, \dots, n\}$  having exactly  $r$  ascents (i.e., places where  $p_i < p_{i+1}$ )

**HarmonicNumber(n)**

The  $n$ th harmonic number  $H_n = \sum_{i=1}^n \frac{1}{i}$ .

**BernoulliNumber(n)**

Returns the  $n$ th Bernoulli number  $B_n$  as a rational number.

**BernoulliApproximation(n)**

Returns a real approximation to the  $n$ th Bernoulli number  $B_n$ .

**BernoulliPolynomial(n)**

The  $n$ th Bernoulli polynomial  $B_n(x) = \sum_{k=0}^n \binom{n}{k} B_k x^{n-k}$  where  $B_n$  is the  $n$ th Bernoulli number.

### 144.3 Subsets of a Finite Set

**Subsets(S)**

The set of all subsets of the set  $S$ .

**Subsets(S, k)**

The set of subsets of the set  $S$  of size  $k$ . If  $k$  is larger than the cardinality of  $S$  then the result will be empty.

**Multisets(S, k)**

The set of multisets consisting of  $k$  not necessarily distinct elements of the set  $S$ .

**Subsequences(S, k)**

The set of sequences of length  $k$  with elements from the set  $S$ .

**Permutations(S)**

The set of permutations (stored as sequences) of the elements of the set  $S$ .

**Permutations(S, k)**

The set of permutations (stored as sequences) of each of the subsets of the set  $S$  of cardinality  $k$ .

---

#### Example H144E1

The use of Subsets is illustrated in the construction of the Petersen graph as the third Odd Graph. The  $n$ th Odd Graph has its vertices in correspondence with the  $(n - 1)$ -element subsets of  $\{1 \dots 2n - 1\}$ , and an edge between two vertices if and only if their corresponding sets have empty intersection.

```
> V := Subsets( {1 .. 2*n-1}, n-1) where n is 3;
> V;
{
  { 1, 5 },
  { 2, 5 },
  { 1, 3 },
  { 1, 4 },
  { 2, 4 },
  { 3, 5 },
  { 2, 3 },
  { 1, 2 },
  { 3, 4 },
  { 4, 5 }
}
> E := { {u, v} : u,v in V | IsDisjoint(u, v) };
> Petersen := Graph< V | E >;
```

---



# 145 PARTITIONS, WORDS AND YOUNG TABLEAUX

<b>145.1 Introduction . . . . .</b>	<b>4813</b>	<b>Content(u)</b>	<b>4820</b>
<b>145.2 Partitions . . . . .</b>	<b>4813</b>	<b>145.4 Tableaux . . . . .</b>	<b>4822</b>
NumberOfPartitions(n)	4813	<i>145.4.1 Tableau Monoids . . . . .</i>	<i>4822</i>
Partitions(n)	4813	TableauMonoid(O)	4822
Partitions(n, k)	4813	TableauMonoid(P)	4822
RestrictedPartitions(n, M)	4813	TableauIntegerMonoid()	4822
RestrictedPartitions(n, k, M)	4813	OrderedMonoid(M)	4823
IsPartition(S)	4813	<i>145.4.2 Creation of Tableaux . . . . .</i>	<i>4824</i>
RandomPartition(n)	4814	Tableau(Q)	4824
Weight(P)	4814	Tableau(Q)	4824
IndexOfPartition(P)	4814	Tableau(S, Q)	4824
<b>145.3 Words . . . . .</b>	<b>4816</b>	Tableau(S, Q)	4824
<i>145.3.1 Ordered Monoids . . . . .</i>	<i>4816</i>	WordToTableau(w)	4824
OrderedMonoid(n)	4816	WordToTableau(u)	4824
OrderedIntegerMonoid()	4816	!	4824
Id(O)	4816	!	4825
!	4816	!	4825
.	4816	!	4825
!	4816	!	4825
!	4817	<i>145.4.3 Enumeration of Tableaux . . . . .</i>	<i>4827</i>
eq	4817	StandardTableaux(P)	4827
*	4817	StandardTableaux(M, P)	4827
IsKnuthEquivalent(w1, w2)	4817	StandardTableauxOfWeight(n)	4827
w[i]	4817	StandardTableauxOfWeight(M, n)	4827
ElementToSequence(w)	4817	TableauxOfShape(S, m)	4827
Eltseq(w)	4817	TableauxOfShape(M, S, m)	4827
Length(w)	4817	TableauxOnShapeWithContent(S, C)	4827
#	4817	TableauxOnShapeWithContent(M, S, C)	4827
Content(w)	4817	TableauxWithContent(C)	4827
IsReverseLatticeWord(w)	4817	TableauxWithContent(M, C)	4827
MaximalIncreasingSequence(w)	4817	<i>145.4.4 Random Tableaux . . . . .</i>	<i>4829</i>
MaximalIncreasingSequences(w, k)	4818	RandomHookWalk(P, i, j)	4829
<i>145.3.2 Plactic Monoids . . . . .</i>	<i>4819</i>	RandomHookWalk(t, i, j)	4829
PlacticMonoid(O)	4819	RandomTableau(S)	4829
PlacticIntegerMonoid()	4819	RandomTableau(M, S)	4829
OrderedMonoid(P)	4819	RandomTableau(n)	4829
Id(P)	4819	RandomTableau(M, n)	4829
!	4819	<i>145.4.5 Basic Access Functions . . . . .</i>	<i>4830</i>
.	4819	Shape(t)	4830
!	4819	OuterShape(t)	4830
!	4819	SkewShape(t)	4830
!	4819	InnerShape(t)	4830
!	4820	PartitionCovers(P1, P2)	4830
!	4820	ConjugatePartition(P)	4830
eq	4820	Weight(t)	4830
*	4820	SkewWeight(t)	4831
Length(u)	4820	NumberOfRows(t)	4831
#	4820	NumberOfSkewRows(t)	4831

Row(t, i)	4831	Rectify(t)	4835
Rows(t)	4831	InverseJeuDeTaquin( $\sim$ t, i, j)	4835
Column(t, j)	4831	InverseJeuDeTaquin(t, i, j)	4835
Columns(t)	4831	RowInsert( $\sim$ t, w)	4836
RowSkewLength(t, i)	4831	RowInsert(t, w)	4836
ColumnSkewLength(t, j)	4831	RowInsert( $\sim$ t, u)	4836
FirstIndexOfRow(t, i)	4831	RowInsert(t, u)	4836
LastIndexOfRow(t, i)	4831	RowInsert( $\sim$ t, x)	4836
RowLength(t, i)	4831	RowInsert(t, x)	4836
FirstIndexOfColumn(t, j)	4832	InverseRowInsert( $\sim$ t, i, j)	4836
LastIndexOfColumn(t, j)	4832	InverseRowInsert(t, i, j)	4836
ColumnLength(t, j)	4832		
145.4.6 Properties . . . . .	4833	145.4.8 The Robinson-Schensted-Knuth Correspondence . . . . .	4838
HookLength(t, i, j)	4833	LexicographicalOrdering( $\sim$ w1, $\sim$ w2)	4838
HookLength(P, i, j)	4833	LexicographicalOrdering(w1, w2)	4838
Content(t)	4833	IsLexicographicallyOrdered(w1, w2)	4838
Word(t)	4833	RSKCorrespondence(w)	4839
RowWord(t)	4833	InverseRSKCorrespondence SingleWord(t1, t2)	4839
ColumnWord(t)	4833	RSKCorrespondence(w1, w2)	4839
IsStandard(t)	4833	InverseRSKCorrespondence DoubleWord(t1, t2)	4839
IsSkew(t)	4833	RSKCorrespondence(M)	4839
IsLittlewoodRichardson(t)	4833	InverseRSKCorrespondence Matrix(t1, t2)	4840
145.4.7 Operations . . . . .	4835	145.4.9 Counting Tableaux . . . . .	4842
eq	4835	NumberOfStandardTableaux(P)	4842
*	4835	NumberOfStandardTableauxOnWeight(n)	4842
DiagonalSum(t1, t2)	4835	NumberOfTableauxOnAlphabet(P, m)	4843
Conjugate(t)	4835	KostkaNumber(S, C)	4843
JeuDeTaquin( $\sim$ t, i, j)	4835		
JeuDeTaquin(t, i, j)	4835	145.5 Bibliography . . . . .	4844
JeuDeTaquin( $\sim$ t)	4835		
JeuDeTaquin(t)	4835		
Rectify( $\sim$ t)	4835		



# Chapter 145

## PARTITIONS, WORDS AND YOUNG TABLEAUX

### 145.1 Introduction

This chapter presents some of the tools provided by MAGMA for partitions, words and Young tableaux.

### 145.2 Partitions

A *partition* of a positive integer  $n$  is a decreasing sequence  $[n_1, n_2, \dots, n_k]$  of positive integers such that  $\sum n_i = n$ . The  $n_i$  are called the *parts* of the partition.

All partition functions in MAGMA operate only on small, positive integers.

NumberOfPartitions(n)

Given a positive integer  $n$ , return the total number of partitions of  $n$ .

Partitions(n)

Given a positive integer  $n$ , return the sequence of all partitions of  $n$ .

Partitions(n, k)

Given positive integers  $n$  and  $k$ , return the sequence of all the partitions of  $n$  into  $k$  parts.

RestrictedPartitions(n, M)

Given a positive integer  $n$  and a set of positive integers  $M$ , return the sequence of all partitions of  $n$ , where the parts are restricted to being elements of the set  $M$ .

RestrictedPartitions(n, k, M)

Given positive integers  $n$  and  $k$ , and a set of positive integers  $M$ , return the sequence of all partitions of  $n$  into  $k$  parts, where the parts are restricted to being elements of the set  $M$ .

IsPartition(S)

A sequence  $S$  is considered to be a partition if it consists of weakly decreasing positive integers. A sequence is allowed to have trailing zeros, and the empty sequence is accepted as a partition (of zero).

**RandomPartition(n)**

Returns a weakly decreasing sequence of positive integers which is random partition of the positive integer  $n$ .

**Weight(P)**

Given a sequence of positive integers  $P$  which is a partition, return a positive integer which is the sum of it's parts.

**IndexOfPartition(P)**

Given a sequence of positive integers  $P$  which is a partition, return its lexicographical order among partitions of the same weight.

Lexicographical ordering of partitions is such that for partitions  $P_1$  and  $P_2$ , then  $P_1 > P_2$  implies that  $P_1$  is greater in the first part which differs from  $P_2$ . The first index is zero.

**Example H145E1**

The conjugacy classes of the symmetric group on  $n$  elements correspond to the partitions of  $n$ . The function **PartnToElt** below converts a partition to an element of the corresponding conjugacy class.

```
> PartitionToElt := function(G, p)
>   x := [];
>   s := 0;
>   for d in p do
>     x cat:= Rotate([s+1 .. s+d], -1);
>     s += d;
>   end for;
>   return G!x;
> end function;
>
> ConjClasses := function(n)
>   G := Sym(n);
>   return [ PartitionToElt(G, p) : p in Partitions(n) ];
> end function;
>
> ConjClasses(5);
[
  (1, 2, 3, 4, 5),
  (1, 2, 3, 4),
  (1, 2, 3)(4, 5),
  (1, 2, 3),
  (1, 2)(3, 4),
  (1, 2),
  Id($)
]
> Classes(Sym(5));
```

## Conjugacy Classes

---

[1]	Order 1	Length 1
	Rep Id(\$)	
[2]	Order 2	Length 10
	Rep (1, 2)	
[3]	Order 2	Length 15
	Rep (1, 2)(3, 4)	
[4]	Order 3	Length 20
	Rep (1, 2, 3)	
[5]	Order 4	Length 30
	Rep (1, 2, 3, 4)	
[6]	Order 5	Length 24
	Rep (1, 2, 3, 4, 5)	
[7]	Order 6	Length 20
	Rep (1, 2, 3)(4, 5)	

**Example H145E2**


---

The number of ways of changing money into five, ten, twenty and fifty cent coins can be calculated using `RestrictedPartitions`. There is also a well known solution using generating functions, which we use as a check.

```
> coins := {5, 10, 20, 50};
> T := [#RestrictedPartitions(n, coins) : n in [0 .. 100 by 5]];
> T;
[ 1, 1, 2, 2, 4, 4, 6, 6, 9, 9, 13, 13, 18, 18, 24, 24, 31, 31, 39, 39, 49 ]
> F<t> := PowerSeriesRing(RationalField(), 101);
> &*[1/(1-t^i) : i in coins];
1 + t^5 + 2*t^10 + 2*t^15 + 4*t^20 + 4*t^25 + 6*t^30 + 6*t^35 + 9*t^40 + 9*t^45
+ 13*t^50 + 13*t^55 + 18*t^60 + 18*t^65 + 24*t^70 + 24*t^75 + 31*t^80 +
31*t^85 + 39*t^90 + 39*t^95 + 49*t^100 + 0(t^101)
```

---

### 145.3 Words

In this and following sections, words are considered to be elements of some *ordered* monoid, that is a free monoid whose generators have an explicit ordering. The most important example is the monoid of words generated by the positive integers, though MAGMA also allows the creation of finitely generated ordered monoids. The words in this section are developed in connection with the theory of Young tableaux (see section 145.4), which are defined over some ordered set of labels (generators of an ordered monoid in MAGMA ).

The importance of the ordering on the generators is in the definition of an equivalence known as *Knuth* equivalence. Knuth equivalence is defined by the two relations

$$\begin{aligned} yxz &\sim yxz & x < y \leq z \\ xzy &\sim zxy & x \leq y < z \end{aligned}$$

The *plactic* monoid is the quotient of an ordered monoid with respect to Knuth equivalence. This plactic monoid is in fact isomorphic to the monoid of tableaux over the same generators. Elements of the plactic monoid are represented by a canonical representative from the Knuth equivalence class of the ordered monoid. This canonical representative is in fact the *row word* of the corresponding tableau (see section 145.4).

#### 145.3.1 Ordered Monoids

An ordered monoid is a free monoid with an ordered basis. MAGMA has two different types of ordered monoids. The first is the monoid of words over the positive integers, which retain their natural ordering. The second are finitely generated monoids, whose generators may be assigned names.

OrderedMonoid(n)
------------------

Given a positive integer  $n$ , return the monoid with  $n$  ordered generators.

OrderedIntegerMonoid()
------------------------

Return the ordered monoid of words over the positive integers.

Id(O)
-------

0 ! 1
-------

Given an ordered monoid  $O$ , return its identity element, i.e., the null word.

0 . i
-------

Given an ordered monoid  $O$  and a positive integer  $i$ , return the  $i$ -th generator of  $O$ .

0 ! [w <sub>1</sub> , ..., w <sub>n</sub> ]
---

Given an ordered monoid  $O$ , and a sequence of elements from  $O$ , return the word  $w_1 \cdots w_n$ .

$O ! [i_1, \dots, i_n]$

Given the ordered monoid  $O$  over the positive integers, and a sequence of integers, return the word  $i_1 \cdots i_n$ .

$w_1 \text{ eq } w_2$

Given two words  $w_1$  and  $w_2$  from the same ordered monoid, return **true** if they are equal.

$w_1 * w_2$

Given two words  $w_1$  and  $w_2$  from the same ordered monoid, return their product under word concatenation.

$\text{IsKnuthEquivalent}(w_1, w_2)$

Two words  $w_1$  and  $w_2$  from the same monoid are Knuth equivalent if they can be transformed into one another using elementary Knuth transformations, (defined in the introduction to this section).

$w[i]$

Given a word  $w$  from an ordered monoid, expressed as a product of generators, return the  $i$ -th generator in the product.

$\text{ElementToSequence}(w)$

$\text{Eltseq}(w)$

Given a word  $w$  from the ordered monoid of positive integers, return  $w$  as a sequence of integers.

$\text{Length}(w)$

$\#w$

Given a word  $w$  from an ordered monoid, return its length.

$\text{Content}(w)$

Given a word  $w$  from an ordered monoid, return a sequence of non-negative integers denoting its content. The content of a word is a sequence where the  $i$ -th position denotes the number of occurrences of the  $i$ -th generator in the word.

$\text{IsReverseLatticeWord}(w)$

A word  $w$  from an ordered monoid is said to be a reverse lattice word (or Yamanouchi word) if for any  $n > 0$ , the last  $n$  letters of  $w$  have a content which is a partition.

$\text{MaximalIncreasingSequence}(w)$

Given a word  $w$  from an ordered monoid, return a weakly increasing subsubsequence of  $w$  of maximal length. This sequence is not necessarily unique.

MaximalIncreasingSequences( $w$ ,  $k$ )

Given a word  $w$  from an ordered monoid and some positive integer  $k$ , return a sequence of  $k$  distinct increasing subsequences of the word  $w$ , such that the maximal number of entries from  $w$  is used. Empty sequences are returned if all entries from  $w$  are used. These sequences are not necessarily unique.

**Example H145E3**

---

We create the ordered monoid over the positive integers and look at a few elements.

```
> O := OrderedIntegerMonoid();
> O;
The monoid of words over the positive integers
>
> w1 := O ! [3,5,2,8];
> w1;
3 5 2 8
> w2 := O ! [6,7,2,4];
> w2;
6 7 2 4
>
> w2[3];
2
> Eltseq(w1);
[ 3, 5, 2, 8 ]
>
> w1*w2;
3 5 2 8 6 7 2 4
```

**Example H145E4**

---

We create a finitely generated ordered monoid and create an element.

```
> O<a,b,c,d> := OrderedMonoid(4);
> O;
The monoid of words over 4 generators: a b c d
>
> w := a*b*a*d*c;
> w;
a b a d c
```

**Example H145E5**

---

We take a word of integers and look at weakly increasing subsequences of maximal length.

```
> O := OrderedIntegerMonoid();
> w := O ! [1,3,4,2,3,4,1,2,2,3,3,2];
>
```

```

> MaximalIncreasingSequence(w);
1 1 2 2 2 3
> MaximalIncreasingSequences(w,2);
[ 1 1 2 2 2 3 , 2 3 3 ]
> MaximalIncreasingSequences(w,3);
[ 1 1 2 2 2 3 , 2 3 3 , 3 4 4 ]
> MaximalIncreasingSequences(w,4);
[ 1 1 2 2 2 3 , 2 3 3 , 3 4 4 , Id(0) ]

```

---

### 145.3.2 Plactic Monoids

The *plactic monoid* of an ordered monoid is the quotient defined by Knuth equivalence. Elements of a plactic monoid are equivalence classes of words from the original ordered monoid, and are represented by a canonical representative.

**PlacticMonoid( $O$ )**

Given an ordered monoid  $O$ , return the plactic monoid obtained by factoring  $O$  by Knuth equivalence.

**PlacticIntegerMonoid()**

Return the plactic monoid obtained by factoring the ordered monoid over the positive integers by Knuth equivalence.

**OrderedMonoid( $P$ )**

Given a plactic monoid  $P$ , return the ordered monoid on which  $P$  is based.

**Id( $P$ )**

**$P ! 1$**

Given a plactic monoid  $P$ , return its identity element which is the null word.

**$P . i$**

Given a plactic monoid  $P$  and a positive integer  $i$ , return the  $i$ -th generator of  $P$ .

**$P ! [u_1, \dots, u_n]$**

Given a plactic monoid  $P$ , and a sequence of elements from  $P$ , return the product  $u_1 \cdots u_n$ .

**$P ! [i_1, \dots, i_n]$**

Given the plactic monoid  $P$  over the positive integers, and a sequence of integers, return the element of  $P$  corresponding to the Knuth equivalence class of  $i_1 \cdots i_n$ .

**$P ! w$**

Given a plactic monoid  $P$ , and a word  $w$  from the ordered monoid that it is based on, return the element of  $P$  corresponding to the Knuth equivalence class of  $w$ .

$P \text{ ! } [w_1, \dots, w_n]$
----------------------------------

Given a plactic monoid  $P$ , and a sequence of elements from the ordered monoid that its based on, return the element of  $P$  corresponding to the Knuth equivalence class of  $w_1 \cdots w_n$ .

$P \text{ ! } t$
------------------

Given a plactic monoid  $P$  and a tableau  $t$  which are both associated with the same ordered monoid, return the element of  $P$  which is uniquely associated to  $t$ .

$u_1 \text{ eq } u_2$
-----------------------

Given two elements  $u_1$  and  $u_2$  from the same plactic monoid, return **true** if they are equal.

$u_1 * u_2$
-------------

Given two elements  $u_1$  and  $u_2$  belonging to the same plactic monoid, return their product which is inherited from word concatenation in the ordered monoid.

$\text{Length}(u)$
--------------------

$\#u$
-------

Given a element  $u$  from a plactic monoid, return its length. The length of a word is invariant under Knuth equivalence and so is well defined for elements of the plactic monoid.

$\text{Content}(u)$
---------------------

Given a word  $u$  from a plactic monoid, return a sequence of non-negative integers denoting its content. The content of a word is a sequence where the  $i$ -th position denotes the number of occurrences of the  $i$ -th generator in the word. The content of a word is invariant under Knuth equivalence and so is well defined for elements of the plactic monoid.

### Example H145E6

---

We create both the ordered monoid and plactic monoid over the integers and look at several elements.

```
> O := OrderedIntegerMonoid();
> P := PlacticIntegerMonoid();
>
> w1 := O ! [2,7,4,8,1,5,9];
> w1;
2 7 4 8 1 5 9
> P!w1;
7 2 8 1 4 5 9
```

First we look at a word that is Knuth equivalent to  $w_1$ ,

```
> w2 := O ! [7,2,1,8,4,5,9];
> w2;
```



```
7 2 1 8 4 5 9
```

```
>
```

```
> IsKnuthEquivalent(w1,w2);
```

```
true
```

```
> (P!w1) eq (P!w2);
```

```
true
```

and then one that is not Knuth equivalent.

```
> w3 := 0 ! [7,1,5,8,2,9,4];
```

```
> w3;
```

```
>
```

```
7 1 5 8 2 9 4
```

```
> IsKnuthEquivalent(w1,w3);
```

```
false
```

```
> (P!w1) eq (P!w3);
```

```
false
```

```
> P!w3;
```

```
7 5 8 1 2 4 9
```

### Example H145E7

---

We create a finitely generated ordered monoid, its associated plactic monoid, and look at some properties which are invariant under Knuth equivalence.

```
> O<a,b,c,d,e> := OrderedMonoid(5);
```

```
> P := PlacticMonoid(O);
```

```
> P;
```

```
The plactic monoid of words over 5 generators: a b c d e
```

```
>
```

```
> w := b*c*e*e*a*d*a*d;
```

```
> w;
```

```
b c e e a d a d
```

```
> Length(w);
```

```
8
```

```
> Content(w);
```

```
[ 2, 1, 1, 2, 2 ]
```

```
>
```

```
> u := P!w;
```

```
> u;
```

```
a a b c d d e e
```

```
> Length(u);
```

```
8
```

```
> Content(u);
```

```
[ 2, 1, 1, 2, 2 ]
```

---

## 145.4 Tableaux

The tableaux module in MAGMA is loosely based on the computational algebra system SYMMETRICA developed by Adalbert Kerber and Axel Kohnert [KKL92]. The main reference for the theory of this section is “Young Tableaux” by William Fulton [Ful97].

A Young diagram, or Ferrers diagram, is a collection of boxes, or cells, arranged in left-justified rows, with a weakly decreasing number of cells in each row. Listing the number of cells in each row gives a partition (its shape) of  $n$ , where  $n$  is the total number of cells in the diagram. Conversely, each partition corresponds to a unique Young diagram. A numbering of a Young diagram is an assignment of a positive integer to each box. More generally a numbering can be done using any ordered set of labels, in MAGMA this means the generators of some ordered monoid, (see section 145.3).

A Young tableau, or simply tableau, is a numbering of a Young diagram which has (i) weakly increasing entries across each row, and (ii) strictly increasing entries down each column. These tableaux form a monoid with respect to a multiplication which may be defined in either of two ways, Schensted’s “bumping” row insertion, or Schützenberger’s “sliding” *jeu de taquin*, (see [Ful97] for full details).

A tableau monoid is constructed from an ordered monoid (see section 145.3), the generators of the ordered monoid comprising the tableau labels. A tableau monoid and a plactic monoid derived from the same ordered monoid are in fact isomorphic to one another.

Flipping a diagram over its main diagonal gives the conjugate diagram, its shape being the `ConjugatePartition` of the original shape.

A skew diagram or skew shape is the diagram obtained by deleting a smaller Young diagram from inside a larger one. A skew tableau is a numbering on a skew diagram obeying the same restrictions on its entries.

### 145.4.1 Tableau Monoids

Let  $O$  be an ordered monoid, elements of  $O$  are expressed as products of its generators. The generators of  $O$  can be used as the set of labels for a family of tableaux  $M$ , called a *tableau monoid*.

A tableau monoid has only one defining characteristic, the ordered monoid which specifies its labels.

The most important tableau monoid is the one defined over the set of all positive integers.

<code>TableauMonoid(O)</code>
-------------------------------

<code>TableauMonoid(P)</code>
-------------------------------

Given an ordered monoid  $O$  or a plactic monoid  $P$ , (in which case let  $O$  be the ordered monoid on which  $P$  is based), then return the tableau monoid of tableaux whose labels are the generators of  $O$ .

<code>TableauIntegerMonoid()</code>
-------------------------------------

Return the tableau monoid over the positive integers.

OrderedMonoid(M)
------------------

Given a tableau monoid  $M$ , return the ordered monoid from which  $M$  gets its labels.

**Example H145E8**

---

We create the tableau monoid over the positive integers. We take a word in the associated ordered monoid and look at its image in the tableau monoid.

```
> M := TableauIntegerMonoid();
> M;
Monoid of Tableaux labelled by the generators of:
The monoid of words over the positive integers
>
> w := OrderedMonoid(M) ! [3,6,2,8,3,9,1];
> w;
3 6 2 8 3 9 1
> M ! w;
Tableau of shape: 4 2 1
1 3 8 9
2 6
3
```

**Example H145E9**

---

We create a tableau monoid over a finite set of labels, and look at the image of a word.

```
> O<x,y,z> := OrderedMonoid(3);
> M := TableauMonoid(O);
> M;
Monoid of Tableaux labelled by the generators of:
The monoid of words over 3 generators: x y z
>
> M ! (z*y*x*z*y*y*x*y*x*z);
Tableau of shape: 5 3 2
x x x y z
y y y
z z
```

---

### 145.4.2 Creation of Tableaux

Tableaux are created from the words of some ordered monoid. For tableaux over the positive integers, the rows can be input directly as sequences. Words can be used to exactly specify the rows of a tableau, or an entire word can be mapped to a tableau using row insertion.

**Tableau( $Q$ )**

Given a sequence of sequences of non-negative integers  $Q$ , return the tableau with the elements of  $Q$  as its rows. Zeroes will indicate skew entries. The resulting tableau must have weakly increasing rows and strictly increasing columns.

**Tableau( $Q$ )**

Given a sequence  $Q$  of words from some ordered monoid, return the (non-skew) tableau with the words of  $Q$  as its rows. The resulting tableau must have weakly increasing rows and strictly increasing columns.

**Tableau( $S, Q$ )**

Given a sequence of nonnegative integers  $S$  which is a partition, and a sequence of sequences of positive integers  $Q$ . Create a skew tableau with skew (inner) shape given by the partition  $S$ , and the non-skew part of each row being given by the entries of  $Q$ . The resulting tableau must have weakly increasing rows and strictly increasing columns.

**Tableau( $S, Q$ )**

Given a sequence of nonnegative integers  $S$  which is a partition, and a sequence  $Q$  of words from some ordered monoid. Create a skew tableau with skew (inner) shape given by the partition  $S$ , and the non-skew part of each row being given by the words of  $Q$ . The resulting tableau must have weakly increasing rows and strictly increasing columns.

**WordToTableau( $w$ )**

**WordToTableau( $u$ )**

Given a word  $w$  from an ordered monoid, return the tableau obtained by row inserting the entries of  $w$  (from the left) into the empty tableau.

If given an element  $u$  of a plactic monoid, any word of its Knuth equivalence class is used. This map is invariant under Knuth equivalence and so is well defined for elements of the plactic monoid.

It is a surjective homomorphism from the ordered monoid to the tableau monoid, and an isomorphism from the plactic monoid to the tableau monoid.

**$M \wr w$**

Given a tableau monoid  $M$  and a word  $w$  from its associated ordered monoid, return the tableau corresponding to  $w$  through row insertion.

M ! u

Given a tableau monoid  $M$  and an element  $u$  from a plactic monoid associated with the same ordered monoid as  $M$ , return the tableau corresponding to  $u$  through row insertion.

M ! [i<sub>1</sub>, ..., i<sub>n</sub>]

Given the tableau monoid  $M$  over the positive integers, and a sequence of positive integers, return the tableau corresponding to the word  $i_1 * \dots * i_n$ .

M ! Q

Given the tableau monoid  $M$  over the positive integers, and a sequence of sequences of non-negative integers  $Q$ , return the tableau with the elements of  $Q$  as its rows. Zeroes will indicate skew entries. The resulting tableau must have weakly increasing rows and strictly increasing columns.

M ! Q

Given a tableau monoid  $M$ , and a sequence  $Q$  of words from its associated ordered monoid, return the (non-skew) tableau with the words of  $Q$  as its rows. The resulting tableau must have weakly increasing rows and strictly increasing columns.

### Example H145E10

---

We create a tableau over the positive integers in two ways. First we input its rows explicitly, then we use a single word which specifies the tableau. We also create a skew tableau by using zero entries in a sequence of sequences.

```
> O := OrderedIntegerMonoid();
>
> T := Tableau( [ [2,5,5,6], [5,7,9,9], [6,9] ] );
> T;
Tableau of shape: 4 4 2
2 5 5 6
5 7 9 9
6 9
> WordToTableau( O ! [6,9,5,7,9,9,2,5,5,6] );
Tableau of shape: 4 4 2
2 5 5 6
5 7 9 9
6 9
> Tableau([ [0,0,0,3,4,5], [0,1,2,4], [1,6] ]);
Skew tableau of shape: 6 4 2 / 3 1
0 0 0 3 4 5
0 1 2 4
1 6
```

**Example H145E11**

---

We create a skew tableau in a tableau monoid with finitely many labels.

```
> O<a,b,c,d> := OrderedMonoid(4);
> T := Tableau( [3,2] , [ a*a*b*d, b*d*d, a*b] );
> T;
Skew tableau of shape: 7 5 2 / 3 2
0 0 0 a a b d
0 0 b d d
a b
```

**Example H145E12**

---

We create the ordered, plactic and tableau monoids each over the positive integers, and examine the natural maps between them.

```
> O := OrderedIntegerMonoid();
> P := PlacticIntegerMonoid();
> M := TableauIntegerMonoid();
> w1 := O ! [4,6,2,6,9,6,2,2,1,7];
> w2 := O ! [9,4,2,1,6,2,6,2,6,7];
> w1 eq w2;
false
> IsKnuthEquivalent(w1,w2);
true
```

Knuth equivalence implies equality for the image in both the plactic and tableau monoids.

```
> (P!w1) eq (P!w2);
true
> P!w1;
9 4 2 6 6 1 2 2 6 7
>
> (M!w1) eq (M!w2);
true
> M!w1;
Tableau of shape: 5 3 1 1
1 2 2 6 7
2 6 6
4
9
```

The plactic and tableau monoids are isomorphic. We confirm that the natural image of an equivalence class from the plactic monoid is the same as the natural images of its member words.

```
> M!w1 eq M ! (P!w1);
true
```

---

### 145.4.3 Enumeration of Tableaux

Multiple tableaux may be created corresponding to specified parameters.

`StandardTableaux(P)`

`StandardTableaux(M, P)`

Returns the set of all standard tableau from the tableau monoid  $M$  of shape  $P$ , which is a partition.

If a tableau monoid  $M$  is not specified then the monoid over the positive integers is used. If  $M$  is specified then it must contain enough labels to fill the shape  $P$ , (i.e. more than  $\text{Weight}(P)$ ).

`StandardTableauxOfWeight(n)`

`StandardTableauxOfWeight(M, n)`

Returns the set of all standard tableau from the tableau monoid  $M$  and of weight  $n$ , which is a positive integer.

If a tableau monoid  $M$  is not specified then the monoid over the positive integers is used. If  $M$  is specified then it must contain at least  $n$  labels.

`TableauxOfShape(S, m)`

`TableauxOfShape(M, S, m)`

Given a tableau monoid  $M$ , a sequence of positive integers  $S$  which is a partition, and a positive integer  $m$ , then return the set of all tableau of shape  $S$  which use only the first  $m$  labels of the tableau monoid  $M$ .

If a tableau monoid  $M$  is not specified then the monoid over the positive integers is used. If  $M$  is specified then it must contain at least  $m$  labels.

`TableauxOnShapeWithContent(S, C)`

`TableauxOnShapeWithContent(M, S, C)`

Given a tableau monoid  $M$ , a sequence of positive integers  $S$  which is a partition, and a sequence of non-negative integers  $C$ , then return the set of all tableau from  $M$  having shape  $S$  and *content*  $C$  (see section 145.4.6 for definition of content). If  $C$  prescribes fewer cells than would completely fill the shape  $S$ , then the tableau within the given shape is returned.

If a tableau monoid  $M$  is not specified then the monoid over the positive integers is used. If  $M$  is specified then it must have at least as many labels as  $C$  has entries.

`TableauxWithContent(C)`

`TableauxWithContent(M, C)`

Given a tableau monoid  $M$ , and a sequence of non-negative integers  $C$ , return the set of all tableau from  $M$  with *content*  $C$  (see section 145.4.6 for definition of content).

If a tableau monoid  $M$  is not specified then the monoid over the positive integers is used. If  $M$  is specified then it must have at least as many labels as  $C$  has entries.

**Example H145E13**

---

We create all tableaux of a given shape, then check that the number of tableaux created agrees with the combinatorical result.

```
> P := [ 3, 2];
> S := StandardTableaux( P );
> S;
{ Tableau of shape: 3 2
  1 2 3
  4 5, Tableau of shape: 3 2
  1 3 4
  2 5, Tableau of shape: 3 2
  1 3 5
  2 4, Tableau of shape: 3 2
  1 2 5
  3 4, Tableau of shape: 3 2
  1 2 4
  3 5 }
>
> #S eq NumberOfStandardTableaux( P );
true
```

**Example H145E14**

---

We create all tableau over the ordered generators  $a, b, c, d$  which have shape  $[4, 2]$  and exactly two  $a$ 's, one  $b$ , two  $c$ 's and one  $d$ . There is in fact four of them.

```
> O<a,b,c,d> := OrderedMonoid(4);
> M := TableauMonoid(0);
> TableauxOnShapeWithContent( M, [4,2], [2,1,2,1]);
{ Tableau of shape: 4 2
  a a c c
  b d , Tableau of shape: 4 2
  a a b c
  c d , Tableau of shape: 4 2
  a a b d
  c c , Tableau of shape: 4 2
  a a c d
  b c }
```

---



### 145.4.4 Random Tableaux

The functions in this section are based on ideas from [GNW84].

RandomHookWalk( $P, i, j$ )
-----------------------------

RandomHookWalk( $t, i, j$ )
-----------------------------

A random hook walk is an essential tool for the creation of random tableau. The *hook* of a cell on a Young diagram is the cells to the right and below its position. The Young diagram is specified by either the sequence of positive integers  $P$  which is a partition, or the tableau  $t$ .

Given positive integers  $i$  and  $j$  such that  $(i, j)$  lies on the shape, the walk proceeds as follows. Starting from the specified  $(i, j)$ -th cell, move to another cell chosen at random from its hook. Repeat this process until a outside corner of the Young diagram is reached, and return the two integers representing the coordinates of this final cell.

RandomTableau( $S$ )
----------------------

RandomTableau( $M, S$ )
-------------------------

Given a tableau monoid  $M$ , and a sequence of positive integers  $S$  which is a partition, return a random standard tableau from  $M$  of shape  $S$ .

If a tableau monoid  $M$  is not specified then the monoid over the positive integers is used. If  $M$  is specified then it must contain enough labels to fill the shape  $S$ , (i.e. more than  $\text{Weight}(S)$ ).

RandomTableau( $n$ )
----------------------

RandomTableau( $M, n$ )
-------------------------

Given a tableau monoid  $M$ , and a positive integer  $n$ , return a (not completely) random standard tableau of weight  $n$ . The probability of any specific tableau of shape  $S$  being returned is  $N_S/n!$ , where  $N_S$  is the number of standard tableaux of shape  $S$ . So tableaux on more populous shapes are more likely to occur.

If a tableau monoid  $M$  is not specified then the monoid over the positive integers is used. If  $M$  is specified then it must contain at least  $n$  labels.

#### Example H145E15

---

We numerically test the random creation of tableau of a given shape, by calculating the average percentage difference in the number of each tableau created. We find it to be less than one percent.

```
> P := [4,3,2];
> Runs := 10000;
>
> S := Setseq( StandardTableaux( P ) );
> Count := [0: i in [1..#S]];
>
> for k in [1..Runs] do
>   T := RandomTableau( P );
```

```

> i := Index(S,T);
> Count[i] += 1;
> end for;
>
> Average := Runs/#S;
> Diff := [RealField(2) | Abs( (Count[i] - Average)/Average )
>           : i in [1..#Count]];
>
> AvgDiff := (&+ Diff)/#Diff;
> AvgDiff;
0.006

```

---

### 145.4.5 Basic Access Functions

**Shape(*t*)**

**OuterShape(*t*)**

The (outer) shape of a tableau  $t$  is the sequence of positive integers denoting the (decreasing) row lengths of its Young Diagram, which results in a partition.

**SkewShape(*t*)**

**InnerShape(*t*)**

The Young diagram of a skew tableau  $t$  has a smaller Young diagram inside of it deleted. The skew, or inner, shape of a skew tableau is the sequence of positive integers denoting the (decreasing) row lengths of this deleted diagram, which results in a partition.

A non-skew tableau is a special case of a skew tableau, whose skew shape is simply the null sequence.

Trailing zeroes are added to the inner shape of  $t$  to give it the same length as the outer shape of  $t$ .

**PartitionCovers(*P1*, *P2*)**

Given two sequences of integers,  $P_1$  and  $P_2$ , which are partitions, return **true** if the Young diagram of  $P_1$  covers the Young diagram of  $P_2$ .

**ConjugatePartition(*P*)**

Given a sequence of integers  $P$  which is a partition, return the partition corresponding to the *conjugate* Young diagram of  $P$  (see section 145.4 for a full description).

**Weight(*t*)**

Given a tableau  $t$ , return the positive integer which is the number of (non-skew) cells in its Young Diagram.

**SkewWeight( $t$ )**

Given a tableau  $t$ , return the positive integer which is the number of skew cells in its Young Diagram.

**NumberOfRows( $t$ )**

Given a tableau  $t$ , return a positive integer which is its number of rows.

**NumberOfSkewRows( $t$ )**

Given a tableau  $t$ , return a positive integer which is its number of skewed rows.

**Row( $t$ ,  $i$ )**

Given a tableau  $t$ , return the non-skewed entries of the  $i$ -th row as a word of an ordered monoid.

**Rows( $t$ )**

Given a tableau  $t$ , return a sequence of words from an ordered monoid which are the non-skewed entries of its rows.

**Column( $t$ ,  $j$ )**

Given a tableau  $t$ , return the non-skewed entries of the  $j$ -th column as a word of an ordered monoid.

**Columns( $t$ )**

Given a tableau  $t$ , return a sequence of words from an ordered monoid which are the non-skewed entries of its columns.

**RowSkewLength( $t$ ,  $i$ )**

Given a tableau  $t$ , return the length of the skewed portion of the  $i$ -th row (zero if no skewed portion).

**ColumnSkewLength( $t$ ,  $j$ )**

Given a tableau  $t$ , return the length of the skewed portion of the  $j$ -th column (zero if no skewed portion).

**FirstIndexOfRow( $t$ ,  $i$ )**

Given a tableau  $t$  and a positive integer  $i$ , return the index of the first non-skew entry of the  $i$ -th row of  $t$ . If the row has no non-skew entries then the index will be one greater than the length of the row, i.e., out of bounds.

**LastIndexOfRow( $t$ ,  $i$ )**

**RowLength( $t$ ,  $i$ )**

Given a tableau  $t$  and a positive integer  $i$ , return the index of the last entry of the  $i$ -th row of  $t$ , which is the length of the  $i$ -th row.

FirstIndexOfColumn( $t$ , $j$ )
---------------------------------

Given a tableau  $t$  and a positive integer  $j$ , return the index of the first non-skew entry of the  $j$ -th column of  $t$ . If the column has no non-skew entries then the index will be one greater than the length of the column, i.e., out of bounds.

LastIndexOfColumn( $t$ , $j$ )
--------------------------------

ColumnLength( $t$ , $j$ )
---------------------------

Given a tableau  $t$  and a positive integer  $j$ , return the index of the last entry in the  $j$ -th column of  $t$ , which is the length of the  $j$ -th column.

---

### Example H145E16

Given a skew tableau, we access a number of its structural properties.

```
> T := Tableau( [3,3,2] , [ [4, 6] , [5, 9], [1, 8] ] );
> T;
Skew tableau of shape: 5 5 4 / 3 3 2
0 0 0 4 6
0 0 0 5 9
0 0 1 8
> Shape(T);
[ 5, 5, 4 ]
> Weight(T);
6
> SkewWeight(T);
8
> NumberOfRows(T);
3
> NumberOfSkewRows(T);
3
> RowSkewLength(T, 2);
3
> Row(T, 2);
5 9
> ColumnSkewLength(T, 3);
2
> Column(T, 3);
1
```

---

### 145.4.6 Properties

HookLength( $t$ , $i$ , $j$ )
-------------------------------

HookLength( $P$ , $i$ , $j$ )
-------------------------------

The *hook* of a cell on a Young diagram comprises the cells to the right and below its position. The Young diagram is specified by either the sequence of positive integers  $P$  which is a partition or the tableau  $t$ . The number of cells contained in a hook is its *length*.

For positive integers  $i$  and  $j$  such that  $(i, j)$  is the co-ordinates of a cell lying on the specified Young diagram, a positive integer is returned which is the length of the hook of that cell.

Content( $t$ )
----------------

Given a tableau  $t$ , return a sequence of non-negative integers which is its *content*.

The content of  $t$  is such that its  $i$ -th value denotes the number of times the  $i$ -th label occurs in  $t$ .

Word( $t$ )
-------------

RowWord( $t$ )
----------------

Given a tableau  $t$ , its row word is formed by reading its entries from left to right, bottom to top. The result is a word of an ordered monoid.

ColumnWord( $t$ )
-------------------

Given a tableau  $t$ , its column word is formed by reading its labels from bottom to top, left to right. The result is a word of an ordered monoid.

IsStandard( $t$ )
-------------------

Given a tableau  $t$  of weight  $n$ , then  $t$  is standard if and only if its entries are exactly the first  $n$  labels of its parent monoid. So  $t$  is standard if and only if it has a content of the form  $[1, 1, \dots, 1]$ .

IsSkew( $t$ )
---------------

Returns `true` if the tableau  $t$  is a skew tableau.

IsLittlewoodRichardson( $t$ )
-------------------------------

Given a tableau  $t$ , then it is a Littlewood–Richardson tableau if the content of  $t$  forms a reverse lattice word, (see section 145.3 for a definition of a reverse lattice word).

**Example H145E17**

---

We create a tableau which is not standard. Examining its content shows a simple way to transform it into a standard tableau.

```
> O := OrderedIntegerMonoid();
> T := WordToTableau( O ! [8,1,7,3,6,2,5,9] );
> T;
Tableau of shape: 4 2 1 1
 1 2 5 9
 3 6
 7
 8
> Content(T);
[ 1, 1, 1, 0, 1, 1, 1, 1, 1 ]
> IsStandard(T);
false
>
> RowInsert(~T, 4);
> T;
Tableau of shape: 4 2 1 1 1
 1 2 4 9
 3 5
 6
 7
 8
> Content(T);
[ 1, 1, 1, 1, 1, 1, 1, 1, 1 ]
> IsStandard(T);
true
```

**Example H145E18**

---

Given a tableau, check that the row and column words are Knuth equivalent.

```
> T := Tableau( [2, 2], [ [1,2,3], [4,6,6], [1,5], [2,6] ] );
> T;
Skew tableau of shape: 5 5 2 2 / 2 2
0 0 1 2 3
0 0 4 6 6
1 5
2 6
>
> RW := RowWord(T);
> RW;
2 6 1 5 4 6 6 1 2 3
> CW := ColumnWord(T);
> CW;
2 1 6 5 4 1 6 2 6 3
```

```
> IsKnuthEquivalent(RW, CW);
true
```

---

### 145.4.7 Operations

Several operations exist to manipulate tableaux and allow them to interact.

`t1 eq t2`

Given two tableaux  $t_1$  and  $t_2$ , return `true` if they are equal.

`t1 * t2`

Given two tableaux  $t_1$  and  $t_2$ , return another tableau which is their product. The product of two tableaux can be defined using either row insertion or *jeu de taquin*.

`DiagonalSum(t1, t2)`

The `DiagonalSum` of two tableau  $t_1$  and  $t_2$  is formed by first building a rectangle of empty cells, with the same number of columns as  $t_1$  and the same number of rows as  $t_2$ . Then  $t_1$  is attached below the rectangle and  $t_2$  attached to right of it.

`Conjugate(t)`

Given a tableau  $t$  with strictly increasing rows, the rows and columns of  $t$  are transposed to form a new tableau on the conjugate Young diagram.

`JeuDeTaquin(~t, i, j)`

`JeuDeTaquin(t, i, j)`

Given a skew tableau  $t$ , and the co-ordinates  $(i, j)$  which must be a skew cell that is an *inside corner* (a corner on the skew shape of  $t$ ). The skewed  $(i, j)$ -th cell is removed using *jeu de taquin*.

`JeuDeTaquin(~t)`

`JeuDeTaquin(t)`

`Rectify(~t)`

`Rectify(t)`

Given a tableau  $t$ , remove all skewed cells successively using *jeu de taquin*.

`InverseJeuDeTaquin(~t, i, j)`

`InverseJeuDeTaquin(t, i, j)`

Given a tableau  $t$ , and co-ordinates  $(i, j)$  which must lie on the outside of  $t$  resulting in a new and valid shape. A new skew tableau with skew degree one higher than  $t$  is created. This is achieved by adding a new skew cell in the  $(i, j)$ -th position, then performing *jeu de taquin* in reverse, which moves the new entry to create a valid skew tableau.

RowInsert( $\sim t$ , $w$ )
-----------------------------

RowInsert( $t$ , $w$ )
------------------------

RowInsert( $\sim t$ , $u$ )
-----------------------------

RowInsert( $t$ , $u$ )
------------------------

Given a tableau  $t$  and a word  $w$  from the same ordered monoid that  $t$  is associated to, insert  $w$  into  $t$  using Schensted's row insertion algorithm.

If given an element  $u$  of a plactic monoid, any word of its Knuth equivalence class is used. This map is invariant under Knuth equivalence and so is well defined for elements of the plactic monoid.

RowInsert( $\sim t$ , $x$ )
-----------------------------

RowInsert( $t$ , $x$ )
------------------------

Given a tableau  $t$  over the positive integers, and a positive integer  $x$ , insert  $x$  into  $t$  using the Schensted's row insertion algorithm.

InverseRowInsert( $\sim t$ , $i$ , $j$ )
--

InverseRowInsert( $t$ , $i$ , $j$ )
-------------------------------------

Given a tableau  $t$  and positive integers  $i$  and  $j$  such that  $(i, j)$  corresponds to an outside corner cell of the tableau  $t$ , then the row insertion algorithm is applied in reverse to remove the specified entry. In the functional form, both the generated tableau and the value of the removed entry as a word of an ordered monoid are returned.

### Example H145E19

---

The order with which the skew entries of a skew tableau are removed using jeu de taquin is irrelevant to the final tableau produced. We give an example of this.

```
> T := Tableau([2,1], [ [2], [3,4] ], [3] );
```

```
> T1 := T;
```

```
> T;
```

```
Skew tableau of shape: 3 3 1 / 2 1
```

```
0 0 2
```

```
0 3 4
```

```
3
```

```
>
```

```
> JeuDeTaquin( $\sim T$ , 1, 2); T;
```

```
Skew tableau of shape: 3 2 1 / 1 1
```

```
0 2 4
```

```
0 3
```

```
3
```

```
> JeuDeTaquin( $\sim T$ , 2, 1); T;
```

```
Skew tableau of shape: 3 2 / 1
```

```
0 2 4
```

```
3 3
```



```

> JeuDeTaquin(~T, 1, 1); T;
Tableau of shape: 3 1
  2 3 4
  3
>
> JeuDeTaquin(~T1, 2, 1); T1;
Skew tableau of shape: 3 3 / 2
  0 0 2
  3 3 4
> JeuDeTaquin(~T1, 1, 2); T1;
Skew tableau of shape: 3 2 / 1
  0 2 4
  3 3
> JeuDeTaquin(~T1, 1, 1); T1;
Tableau of shape: 3 1
  2 3 4
  3
>
> T eq T1;
true

```

### Example H145E20

---

We manually compare the two different methods of multiplying tableau.

```

> O<a,b,c,d,e,f,g,h> := OrderedMonoid(8);
> T1 := Tableau( [ a*c*f, d*g] );
> T2 := Tableau( [ b*b*e*f, d*g*h] );

```

First using row insert,

```

> w := Word(T2);
> w;
d g h b b e f
> Res1 := RowInsert(T1, w);
> Res1;
Tableau of shape: 5 4 2 1
a b b e f
c d g h
d f
g

```

and then using JeuDeTaquin.

```

> Res2 := DiagonalSum(T1, T2);
> Res2;
Skew tableau of shape: 7 6 3 2 / 3 3
0 0 0 b b e f
0 0 0 d g h
a c f

```

```

d g
> Rectify(~Res2);
> Res2;
Tableau of shape: 5 4 2 1
a b b e f
c d g h
d f
g

```

Now we check our results.

```

> Res1 eq Res2;
true
> Res1 eq (T1*T2);
true

```

#### 145.4.8 The Robinson-Schensted-Knuth Correspondence

While the map from the plactic monoid to tableaux is an isomorphism, many different words from the original ordered monoid map to the same tableau. However a bijective map exists between an ordered monoid and *pairs* of tableau of the same shape, the second of which being a standard tableau. This is known as the *Robinson correspondence*.

By removing the restriction that the second tableau be standard, this correspondence is extended further to a bijective map with all matrices with non-negative integers. This is known as the Robinson-Schensted-Knuth (RSK) correspondence.

This latter correspondence can also be made to all lexicographically ordered pairs of words (both having the same length).

LexicographicalOrdering(~w1, ~w2)

LexicographicalOrdering(w1, w2)

Given two words  $w_1$  and  $w_2$  of the same length, place them in increasing lexicographical order (the functional version returns two new words). Lexicographical ordering on the pairs  $(w_1[i], w_2[i])$  is determined primarily by the comparison  $w_2[i] < w_2[j]$ , but if  $w_2[i] = w_2[j]$  then the comparison  $w_1[i] < w_1[j]$  is taken into account.

IsLexicographicallyOrdered(w1, w2)

Given two words  $w_1$  and  $w_2$ , return `true` if the pairs  $(w_1[i], w_2[i])$  are ordered according to the ordering described in `LexicographicalOrdering`.

**RSKCorrespondence(w)**

Given a word  $w$ , use the bijective map described by the Robinson-Schensted-Knuth correspondence to return two tableaux  $t_1$  and  $t_2$ . The first tableau,  $t_1$ , will be labelled by the entries of  $w$ , while  $t_2$  is a standard tableau over the positive integers.

The map described by the correspondence is as follows. Map the word  $w$  to  $t_1$  by row inserting its entries into the empty tableau, (as is done in `WordToTableau`). As  $t_1$  is being generated, the insertion tableau  $t_2$  is built simultaneously. As  $w[k]$  is inserted into  $t_1$ , an additional cell is added to its shape. A new cell is added to  $t_2$  in exactly the same place, and it is labelled with the integer  $k$ .

**InverseRSKCorrespondenceSingleWord(t1, t2)**

Given a pair of tableaux  $t_1, t_2$ , of the same shape, where  $t_2$  is over the positive integers, return the preimage of  $t_1$  and  $t_2$  using the bijective map described by the Robinson-Schensted-Knuth correspondence.

The result is returned as a single word of the ordered monoid of  $t_1$ , as described in the single word version of `RSKCorrespondence`.

**RSKCorrespondence(w1, w2)**

Given two lexicographically ordered words  $w_1$  and  $w_2$  of the same length, where  $w_2$  must be over the positive integers, use the bijective map described by the Robinson-Schensted-Knuth correspondence to return two tableaux  $t_1$  and  $t_2$ . The first tableau,  $t_1$ , will be labelled by the entries of  $w_1$ , while  $t_2$  will be labelled by the entries of  $w_2$ .

The map described by the correspondence is as follows. Map the word  $w_1$  to  $t_1$  by row inserting its entries into the empty tableau, (as is done in `WordToTableau`). As  $t_1$  is being generated, the insertion tableau  $t_2$  is built simultaneously. As  $w_1[k]$  is inserted into  $t_1$ , an additional cell is added to its shape. A new cell is added to  $t_2$  in exactly the same place, and it is labelled with the integer  $w_2[k]$ .

**InverseRSKCorrespondenceDoubleWord(t1, t2)**

Given a pair of tableaux  $t_1, t_2$ , of the same shape, where  $t_2$  is over the positive integers, return the preimage of  $t_1$  and  $t_2$  using the bijective map described by the Robinson-Schensted-Knuth correspondence.

The result is returned as words of the ordered monoids of  $t_1$  and  $t_2$  respectively, as described in the double word version of `RSKCorrespondence`.

**RSKCorrespondence(M)**

Given a matrix  $M$  of non-negative integers, use the bijective map described by the Robinson-Schensted-Knuth correspondence to return two tableaux  $t_1$  and  $t_2$  over the positive integers.

The map described by the correspondence is as follows. Initialise two words over the positive integers,  $w_1$  and  $w_2$ , as null words. For the  $(i, j)$ -th entry of  $M$ , say  $m_{ij}$ , append  $i$  to  $w_1$   $m_{ij}$  times, and append  $j$  to  $w_2$   $m_{ij}$  times. These two words are then used in the correspondence described in the above version of `RSKCorrespondence`.

## InverseRSKCorrespondenceMatrix(t1, t2)

Given a pair of tableaux  $t_1, t_2$ , of the same shape, both over the positive integers, return the preimage of  $t_1$  and  $t_2$  using the bijective map described by the Robinson-Schensted-Knuth correspondence.

The result is a matrix of non-negative integers, as described in the matrix version of RSKCorrespondence.

**Example H145E21**

---

We take two words from the ordered monoid over the positive integers. Although they correspond to the same tableau under the natural mapping into the tableau monoid, we show they have a different image under the RSK correspondence. That is, they have different *insertion* tableaux.

```
> O := OrderedIntegerMonoid();
> M := TableauMonoid(O);
> a := 0 ! [4,7,2,8,4,9,2];
> a;
4 7 2 8 4 9 2
> b := 0 ! [7,4,2,4,2,8,9];
> b;
7 4 2 4 2 8 9
```

Now we will see that  $a$  and  $b$  have the same image in the tableau monoid, but that they are distinguished under the RSK correspondence.

```
> (M!a) eq (M!b);
true
> Ta1, Ta2 := RSKCorrespondence(a);
> Tb1, Tb2 := RSKCorrespondence(b);
> Ta1 eq Tb1;
true
> Ta2 eq Tb2;
false
> Ta2;
Tableau of shape: 4 2 1
1 2 4 6
3 5
7
> Tb2;
Tableau of shape: 4 2 1
1 4 6 7
2 5
3
```

To check that the map is indeed bijective we find  $a$  and  $b$  as the pre-images of the created tableaux.

```
> InverseRSKCorrespondenceSingleWord(Ta1,Ta2 );
4 7 2 8 4 9 2
> InverseRSKCorrespondenceSingleWord(Tb1,Tb2 );
```

7 4 2 4 2 8 9

### Example H145E22

---

We apply the RSK correspondence to two arbitrary words, the first from a finitely generated ordered monoid. The second word must always be over the positive integers. The first step is to put them into lexicographical order.

```
> O1<a,b,c,d,e,f> := OrderedMonoid(6);
> O2 := OrderedIntegerMonoid();
>
> w1 := e*a*e*b*f*d*a;
> w2 := O2 ! [3,8,2,8,3,3,6];
> LexicographicalOrdering(~w1, ~w2);
> w1, w2;
e d e f a a b 2 3 3 3 6 8 8
```

Now we use the correspondence,

```
> T1, T2 := RSKCorrespondence(w1, w2);
> T1;
Tableau of shape: 3 3 1
a a b
d e f
e
> T2;
Tableau of shape: 3 3 1
2 3 3
3 8 8
6
```

and check that the inverse is correct.

```
> InverseRSKCorrespondenceDoubleWord(T1, T2);
e d e f a a b 2 3 3 3 6 8 8
```

### Example H145E23

---

We give an example of the bijective map provided by the RSK correspondence between matrices of non-negative integers and pairs of tableaux of the same shape.

```
> M := Matrix(2,3,[0,0,2,3,1,2]);
> M;
[0 0 2]
[3 1 2]
> T1, T2 := RSKCorrespondence(M);
> T1;
Tableau of shape: 6 2
1 1 1 2 3 3
3 3
```

```

> T2;
Tableau of shape: 6 2
1 1 2 2 2 2
2 2
> InverseRSKCorrespondenceMatrix(T1, T2);
[0 0 2]
[3 1 2]

```

Looking now at preimage of the tableaux in the double word format, the  $(i, j)$  co-ordinates of entries of  $M$  can clearly be seen with the correct multiplicities.

```

> wj, wi := InverseRSKCorrespondenceDoubleWord(T1, T2);
> wi;
1 1 2 2 2 2 2 2
> wj;
3 3 1 1 1 2 3 3

```

---

#### Example H145E24

We illustrate the remarkable property that a permutation  $p$  (in image notation) corresponds to the tableaux pair  $(T1, T2)$  if and only if the inverse permutation  $p^{-1}$  corresponds to the tableaux pair  $(T2, T1)$ .

```

> O := OrderedIntegerMonoid();
> n := Random([1..100]);
> G := SymmetricGroup(n);
> p := Random(G);
>
> T1, T2 := RSKCorrespondence( O ! Eltseq(p) );
>
> p1 := InverseRSKCorrespondenceSingleWord( T2, T1);
> p1 := G ! Eltseq(p1);
>
> p1 eq p^-1;
true

```

---

### 145.4.9 Counting Tableaux

**NumberOfStandardTableaux(P)**

Given a sequence of positive integers  $P$  which is a partition, return the number of standard tableaux of shape  $P$ . This is the same as the number of Knuth equivalent words corresponding to each standard tableaux of shape  $P$ .

**NumberOfStandardTableauxOnWeight(n)**

Given a positive integer  $n$ , return the number of standard tableaux having weight  $n$ .

NumberOfTableauxOnAlphabet( $P$ , $m$ )
---

Given a sequence of positive integers  $P$  which is a partition, return the number of tableau of shape  $P$  with entries from  $[1, \dots, m]$ .

KostkaNumber( $S$ , $C$ )
---------------------------

Given a sequence of positive integers  $S$  forming a partition, and a sequence of non-negative integers  $C$ , return the number of tableau having shape  $S$  and content  $C$ . If  $C$  prescribes fewer cells than would completely fill the shape  $S$ , then the number of tableau within the given shape is returned.

---

**Example H145E25**

There is a correspondence between the number of standard tableaux on a given shape, and the number of words associated with each of those tableaux. We manually count the number of words corresponding to a specific standard tableau, and show that this is equal to the number of standard tableaux on that shape.

```
> O := OrderedIntegerMonoid();
> T := Tableau( [ [ 1, 2, 3, 4, 7],
>                [ 5, 8],
>                [ 6]   ] );
> T;
Tableau of shape: 5 2 1
 1 2 3 4 7
 5 8
 6
>
> n := Weight(T);
> G := SymmetricGroup(n);
> S := [1..n];
>
> Count := 0;
> for p in G do
>   T1 := WordToTableau( O ! (S^p) );
>   if T1 eq T then
>     Count += 1;
>   end if;
> end for;
>
> Count;
64
> NumberOfStandardTableaux( Shape(T) );
64
```

**Example H145E26**

---

If a tableaux is constructed of the shape  $[1, 1, 1, \dots, 1]$  then the entries must be strictly increasing, hence non-repeating. So for such a tableau of length  $n$  on an alphabet of  $m$  symbols, each subset of  $n$  symbols will correspond to one single distinct tableau. So the number of tableau will be  $\binom{m}{n}$ .

```
> m := Random(1,50);
> n := Random(1,m);
>
> Shape := [ 1 : i in [1..n] ];
> Binomial(m,n) eq NumberOfTableauxOnAlphabet(Shape, m);
true
```

---

**145.5 Bibliography**

- [Ful97] William Fulton. *Young Tableaux*. Cambridge University Press, Cambridge, 1997.
- [GNW84] Curtis Greene, Albert Nijenhuis, and Herbert S. Wilf. Another probabilistic method in the theory of Young tableaux. *J. Combin. Theory Ser. A*, 37(2):127–135, 1984.
- [KKL92] Adalbert Kerber, Axel Kohnert, and Alain Lascoux. SYMMETRICA, an object oriented computer-algebra system for the symmetric group. *J. Symbolic Comp.*, 14(2-3):195–203, 1992.  
URL:<http://www.mathe2.uni-bayreuth.de/axel/symneu-engl.html>.



# 146 SYMMETRIC FUNCTIONS

<b>146.1 Introduction . . . . .</b>	<b>4847</b>	<b>A ‘ PrintStyle . . . . .</b>	<b>4856</b>
<b>146.2 Creation . . . . .</b>	<b>4849</b>	<b>146.4.3 Additive Arithmetic Operators . . . . .</b>	<b>4856</b>
146.2.1 Creation of Symmetric Function		+ - . . . . .	4857
Algebras . . . . .	4849	+ - . . . . .	4857
SymmetricFunctionAlgebra(R)	4850	+= -= . . . . .	4857
SFA(R)	4850	146.4.4 Multiplication . . . . .	4857
SymmetricFunctionAlgebraSchur(R)	4850	* . . . . .	4857
SFASchur(R)	4850	*:= . . . . .	4857
SymmetricFunctionAlgebra		^ . . . . .	4857
Homogeneous . . . . .	4850	146.4.5 Plethysm . . . . .	4858
SFAHomogeneous(R)	4850	~ . . . . .	4858
SymmetricFunctionAlgebraPower(R)	4850	146.4.6 Boolean Operators . . . . .	4858
SFAPower(R)	4850	IsHomogeneous(s) . . . . .	4858
SymmetricFunctionAlgebraElementary(R)	4850	IsZero IsOne IsMinusOne . . . . .	4858
SFAElementary(R)	4850	eq . . . . .	4858
SymmetricFunctionAlgebraMonomial(R)	4850	ne . . . . .	4858
SFAMonomial(R)	4850	146.4.7 Accessing Elements . . . . .	4859
146.2.2 Creation of Symmetric Functions	4851	Coefficient(s, p) . . . . .	4859
. . . . .	4851	Support(s) . . . . .	4859
. . . . .	4851	Length(s) . . . . .	4859
IsCoercible(A, f) . . . . .	4852	Degree(s) . . . . .	4859
! . . . . .	4852	146.4.8 Multivariate Polynomials . . . . .	4860
! . . . . .	4852	! . . . . .	4860
! . . . . .	4853	146.4.9 Frobenius Homomorphism . . . . .	4861
<b>146.3 Structure Operations . . . . .</b>	<b>4854</b>	Frobenius(s) . . . . .	4861
146.3.1 Related Structures . . . . .	4854	146.4.10 Inner Product . . . . .	4862
BaseRing(L) . . . . .	4854	InnerProduct(a,b) . . . . .	4862
CoefficientRing(L) . . . . .	4854	146.4.11 Combinatorial Objects . . . . .	4862
Category Parent PrimeRing . . . . .	4854	Tableaux(sf, m) . . . . .	4862
146.3.2 Ring Predicates and Booleans . . . . .	4855	146.4.12 Symmetric Group Character . . . . .	4862
IsCommutative IsUnitary . . . . .	4855	SymmetricCharacter(sf) . . . . .	4862
IsFinite IsOrdered . . . . .	4855	146.4.13 Restrictions . . . . .	4863
IsField IsEuclideanDomain . . . . .	4855	RestrictDegree(a, n) . . . . .	4863
IsPID IsUFD . . . . .	4855	RestrictPartitionLength(a, n) . . . . .	4863
IsDivisionRing IsEuclideanRing . . . . .	4855	RestrictParts(a, n) . . . . .	4863
IsDomain . . . . .	4855	<b>146.5 Transition Matrices . . . . .</b>	<b>4864</b>
IsPrincipalIdealRing . . . . .	4855	146.5.1 Transition Matrices from Schur Basis . . . . .	4864
eq . . . . .	4855	SchurToMonomialMatrix(n) . . . . .	4864
ne . . . . .	4855	SchurToHomogeneousMatrix(n) . . . . .	4865
146.3.3 Predicates on Basis Types . . . . .	4855	SchurToPowerSumMatrix(n) . . . . .	4865
HasSchurBasis(A) . . . . .	4855	SchurToElementaryMatrix(n) . . . . .	4865
HasHomogeneousBasis(A) . . . . .	4855	146.5.2 Transition Matrices from Monomial Basis . . . . .	4866
HasElementaryBasis(A) . . . . .	4855	MonomialToSchurMatrix(n) . . . . .	4866
HasPowerSumBasis(A) . . . . .	4855	MonomialToHomogeneousMatrix(n) . . . . .	4866
HasMonomialBasis(A) . . . . .	4855	MonomialToPowerSumMatrix(n) . . . . .	4866
<b>146.4 Element Operations . . . . .</b>	<b>4855</b>		
146.4.1 Parent and Category . . . . .	4855		
Parent Category . . . . .	4855		
146.4.2 Print Styles . . . . .	4856		

MonomialToElementaryMatrix(n)	4866	PowerSumToMonomialMatrix(n)	4869
146.5.3 Transition Matrices from Homogeneous Basis . . . . .	4867	PowerSumToHomogeneousMatrix(n)	4869
HomogeneousToSchurMatrix(n)	4867	PowerSumToElementaryMatrix(n)	4869
HomogeneousToMonomialMatrix(n)	4867	146.5.5 Transition Matrices from Elementary Basis . . . . .	4869
HomogeneousToPowerSumMatrix(n)	4867	ElementaryToSchurMatrix(n)	4869
HomogeneousToElementaryMatrix(n)	4867	ElementaryToMonomialMatrix(n)	4869
146.5.4 Transition Matrices from Power Sum Basis . . . . .	4868	ElementaryToHomogeneousMatrix(n)	4870
PowerSumToSchurMatrix(n)	4868	ElementaryToPowerSumMatrix(n)	4870
		<b>146.6 Bibliography . . . . .</b>	<b>4870</b>

# Chapter 146

## SYMMETRIC FUNCTIONS

### 146.1 Introduction

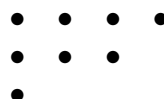
A *symmetric function* is a polynomial which is invariant under permutations of its indeterminates. The symmetric functions over a commutative ring with unity, with an arbitrary number of indeterminates, form an algebra, denoted by  $\Lambda$ .

The symmetric functions of fixed degree  $n$  form a submodule of  $\Lambda$  denoted by  $\Lambda^n$ . When analysing and computing with elements of  $\Lambda$ , there are 5 important bases to consider. These are the bases consisting of the *Schur*, *Homogeneous*, *Power Sum*, *Elementary* and *Monomial* symmetric functions. The size of any basis of  $\Lambda^n$  is equal to the number of partitions of weight  $n$ , and each basis has its elements indexed by those partitions.

A *partition* is a weakly decreasing sequence of positive integers. So, for example,  $[5, 3, 1]$  is a partition of weight 9, and  $[3, 1]$  is a partition of weight 4. For more information on partitions see Section 145.2.

MAGMA allows computations with symmetric functions in any of the 5 bases, and any symmetric function can be expressed in any of the different bases. The main reference on the theory of symmetric functions used here is the book of Macdonald [Mac95].

In order to understand the different symmetric function bases, it is necessary to have an understanding of *Young Tableaux*. Firstly, for a given partition let us visualise its *shape* making a diagram. We arrange “boxes” in left-justified rows corresponding to the entries of the partition. So for example, the partition  $[4, 3, 1]$  of weight 8 gives



This is known as a *Ferrers diagram*.

Now, to create a Young tableau, we must fill these boxes with integer entries, but subject to some restrictions. We require that each row of the tableau must be weakly increasing, and each column must be strictly increasing. So, for example, a possible tableau of shape  $[4, 3, 1]$  would be

$$T = \begin{array}{cccc} 1 & 3 & 3 & 4 \\ 2 & 4 & 5 & \\ 4 & & & \end{array}$$

A tableau can be mapped onto a monomial by treating each entry in the tableau as an indeterminate. So, the above tableau will correspond to the monomial

$$T \mapsto x^T = x_1 x_2 x_3^2 x_4^3 x_5$$

With this connection between tableaux and monomials we are now able to define the bases of an algebra of symmetric functions.

An important point to realise is that when dealing with a symmetric function, the number of indeterminates of the corresponding polynomial is neither fixed nor important. In fact a symmetric function has an image as a symmetric polynomial in an arbitrary number of indeterminates, even an infinite number. MAGMA is only equipped to deal with polynomial rings with finite rank.

Since the partitions of weight  $n$  prescribe the symmetric function basis elements of degree  $n$  which generate  $\Lambda^n$ , (the symmetric polynomials in  $n$  indeterminates), then the number of indeterminates being used will normally be equal to the degree of the symmetric functions considered.

- Schur Functions

Given a partition  $\lambda = [\lambda_1, \lambda_2, \dots, \lambda_l]$  of weight  $n$ , the Schur function  $s_\lambda$  is given by  $s_\lambda = \sum_T x^T$  where the sum is taken over all Young tableaux with shape  $\lambda$ , whose entries are all less than or equal to  $m$ , (the number of required indeterminates).

- Elementary Functions

For a positive integer  $k$ , define the  $k$ -th elementary symmetric function,  $e_k$ , as the Schur function of  $[1^k]$ . Then, for any partition  $\lambda = [\lambda_1, \lambda_2, \dots, \lambda_l]$ , we define the elementary symmetric function indexed by  $\lambda$  to be  $e_\lambda = \prod_i e_{\lambda_i}$ .

- Homogeneous Functions

For a positive integer  $k$ , define the  $k$ -th homogeneous symmetric function,  $h_k$ , as the Schur function of  $[k]$ . Then, for any partition  $\lambda = [\lambda_1, \lambda_2, \dots, \lambda_l]$ , we define the homogeneous symmetric function indexed by  $\lambda$  to be  $h_\lambda = \prod_i h_{\lambda_i}$ .

- Monomial Functions

For a partition  $\lambda = [\lambda_1, \lambda_2, \dots, \lambda_l]$ , let  $x^\lambda$  be given by  $x_1^{\lambda_1} x_2^{\lambda_2} \dots x_l^{\lambda_l}$ . The monomial symmetric function  $m_\lambda$  is the sum over the orbits of  $x^\lambda$  under the action of the symmetric group. Where the number of indeterminates exceeds the length of the partition,  $\lambda$  can be considered to have trailing zeros.

- Power Sums

For a positive integer  $k$ , define the  $k$ -th power sum symmetric function,  $p_k$ , as the monomial function of  $[k]$ . Then, for any partition  $\lambda = [\lambda_1, \lambda_2, \dots, \lambda_l]$ , we define the power sum symmetric function indexed by  $\lambda$  to be  $p_\lambda = \prod_i p_{\lambda_i}$ .

For each of these classes of functions, the set of all functions indexed by partitions of weight  $n$  forms a basis for  $\Lambda^n$ . Taking functions indexed by partitions of *all* weights gives a basis of  $\Lambda$ .

It is possible to define an inner product on  $\Lambda$  by requiring that the bases  $m_\lambda$  and  $h_\lambda$  be dual to each other:

$$\langle m_\lambda, h_{\lambda'} \rangle = \delta_{\lambda, \lambda'}$$

With respect to this inner product, the Schur functions are an orthonormal basis of  $\Lambda^n$  and  $\Lambda$  and the power sum basis is an orthogonal basis.

---

### Example H146E1

We give an example illustrating the correspondence between symmetric functions written with respect to a symmetric function basis and symmetric polynomials. While the degree of the sym-

metric function fixes the degree of the resulting polynomial, the number of indeterminates is arbitrary.

```

> Q := Rational();
> S := SFASchur(Rational());
> e := S.[2,1];
> P2<[x]> := PolynomialRing(Q, 2);
> P2 ! e;
x[1]^2*x[2] + x[1]*x[2]^2
> P3<[y]> := PolynomialRing(Q, 3);
> P3 ! e;
y[1]^2*y[2] + y[1]^2*y[3] + y[1]*y[2]^2 + 2*y[1]*y[2]*y[3] + y[1]*y[3]^2 +
  y[2]^2*y[3] + y[2]*y[3]^2
> P4<[z]> := PolynomialRing(Q, 4);
> P4 ! e;
z[1]^2*z[2] + z[1]^2*z[3] + z[1]^2*z[4] + z[1]*z[2]^2 + 2*z[1]*z[2]*z[3] +
  2*z[1]*z[2]*z[4] + z[1]*z[3]^2 + 2*z[1]*z[3]*z[4] + z[1]*z[4]^2 +
  z[2]^2*z[3] + z[2]^2*z[4] + z[2]*z[3]^2 + 2*z[2]*z[3]*z[4] +
  z[2]*z[4]^2 + z[3]^2*z[4] + z[3]*z[4]^2
>
> P5<[w]> := PolynomialRing(Q, 5);
> P5 ! e;
w[1]^2*w[2] + w[1]^2*w[3] + w[1]^2*w[4] + w[1]^2*w[5] + w[1]*w[2]^2 +
  2*w[1]*w[2]*w[3] + 2*w[1]*w[2]*w[4] + 2*w[1]*w[2]*w[5] + w[1]*w[3]^2 +
  2*w[1]*w[3]*w[4] + 2*w[1]*w[3]*w[5] + w[1]*w[4]^2 + 2*w[1]*w[4]*w[5] +
  w[1]*w[5]^2 + w[2]^2*w[3] + w[2]^2*w[4] + w[2]^2*w[5] + w[2]*w[3]^2 +
  2*w[2]*w[3]*w[4] + 2*w[2]*w[3]*w[5] + w[2]*w[4]^2 + 2*w[2]*w[4]*w[5] +
  w[2]*w[5]^2 + w[3]^2*w[4] + w[3]^2*w[5] + w[3]*w[4]^2 +
  2*w[3]*w[4]*w[5] + w[3]*w[5]^2 + w[4]^2*w[5] + w[4]*w[5]^2

```

---

## 146.2 Creation

### 146.2.1 Creation of Symmetric Function Algebras

Symmetric functions are symmetric polynomials over some coefficient ring, hence the algebra of symmetric functions is defined by specifying this ring. There are five standard bases that can be used to express symmetric functions. In MAGMA there are separate constructions for creating an algebra using each distinct basis.

SymmetricFunctionAlgebra(R)

SFA(R)

Basis

MONSTGELT

Default : “*Schur*”

Given a ring  $R$ , return the algebra of symmetric functions over  $R$ . By default this algebra will use the basis of “**Schur**” functions, though the parameter **Basis** allows the user to specify that the basis be one of the “**Homogeneous**”, “**PowerSum**”, “**Elementary**” or “**Monomial**” functions.

SymmetricFunctionAlgebraSchur(R)

SFASchur(R)

Given a commutative ring with unity  $R$ , create the algebra of symmetric functions (polynomials) with coefficients from  $R$ . Its elements are expressed in terms of the basis of *Schur* symmetric functions, which are indexed by partitions.

SymmetricFunctionAlgebraHomogeneous

SFAHomogeneous(R)

Given a commutative ring with unity  $R$ , create the algebra of symmetric functions (polynomials) with coefficients from  $R$ . Its elements are expressed in terms of the basis of *Homogeneous* symmetric functions, which are indexed by partitions.

SymmetricFunctionAlgebraPower(R)

SFAPower(R)

Given a commutative ring with unity  $R$ , create the algebra of symmetric functions (polynomials) with coefficients from  $R$ . Its elements are expressed in terms of the basis of *Power Sum* symmetric functions, which are indexed by partitions.

SymmetricFunctionAlgebraElementary(R)

SFAElementary(R)

Given a commutative ring with unity  $R$ , create the algebra of symmetric functions (polynomials) with coefficients from  $R$ . Its elements are expressed in terms of the basis of *Elementary* symmetric functions, which are indexed by partitions.

SymmetricFunctionAlgebraMonomial(R)

SFAMonomial(R)

Given a commutative ring with unity  $R$ , create the algebra of symmetric functions (polynomials) with coefficients from  $R$ . Its elements are expressed in terms of the basis of *Monomial* symmetric functions, which are indexed by partitions.

**Example H146E2**

---

A symmetric polynomial can be expressed in terms of any of the 5 standard bases.

```
> R := Rational();
> S := SymmetricFunctionAlgebraSchur(R);
Symmetric Algebra over Rational Field, Schur symmetric functions as basis
> E := SymmetricFunctionAlgebra(R : Basis := "Elementary");
Symmetric Algebra over Rational Field, Elementary symmetric functions as basis
> f1 := S.[2,1];
> f1;
S.[2,1]
>
> f2 := E ! f1;
> f2;
E.[2,1] - E.[3]
> f1 eq f2;
true
>
> P<x,y,z> := PolynomialRing(R, 3);
> P ! f1;
x^2*y + x^2*z + x*y^2 + 2*x*y*z + x*z^2 + y^2*z + y*z^2
> P ! f2;
x^2*y + x^2*z + x*y^2 + 2*x*y*z + x*z^2 + y^2*z + y*z^2
```

---

**146.2.2 Creation of Symmetric Functions**

For each of the 5 different standard basis for symmetric functions, basis elements are indexed via partitions (weakly decreasing positive sequences). The *weight* of a partition is the sum of its entries, and gives the degree of the element.

For example,  $[3, 1]$  is a partition of weight 4, hence a symmetric function basis element corresponding to  $[3, 1]$  will be a symmetric polynomial of degree 4.

General symmetric functions are linear combinations of basis elements and can be created by taking such linear combinations or via coercion from either another basis or directly from a polynomial.

<b>A . P</b>
--------------

Given a partition  $P$ , which is a weakly decreasing positive sequence of integers, return the basis element of the algebra of symmetric functions  $A$  corresponding to  $P$ .

<b>A . i</b>
--------------

Given a positive integer  $i$ , return the basis element of the algebra of symmetric functions  $A$  corresponding to the partition  $[i]$ .

**Example H146E3**

---

We can create elements via linear combinations of basis elements.

```
> R := Rational();
> M := SFAMonomial(R);
> M.[3,1];
M.[3,1]
> M.4;
M.[4]
> 3 * M.[3,1] - 1/2 * M.4;
3*M.[3,1] - 1/2*M.[4]
```

---

IsCoercible(A, f)
-------------------

A ! f
-------

Given a multivariate polynomial  $f$  which is symmetric in all of its indeterminates (and hence is a symmetric function), return  $f$  as an element of the algebra of symmetric functions  $A$ . This element returned will be expressed in terms of the symmetric function basis of  $A$ .

**Example H146E4**

---

Symmetric polynomials are symmetric functions. These can be coerced into algebras of symmetric functions and hence expressed in terms of the relevant basis. Polynomials which are not symmetric cannot be coerced.

```
> R := Rational();
> M := SFAMonomial(R);
> P<[x]> := PolynomialRing(R, 3);
> f := -3*x[1]^3 + x[1]^2*x[2] + x[1]^2*x[3] + x[1]*x[2]^2 + x[1]*x[3]^2 -
>      3*x[2]^3 + x[2]^2*x[3] + x[2]*x[3]^2 - 3*x[3]^3;
> M!f;
M.[2,1] - 3*M.[3]
> M ! (x[1] + x[2]*x[3]);
>> M ! (x[1] + x[2]*x[3]);
^
```

Runtime error in '!': Polynomial is not symmetric

---

A ! r
-------

Create the scalar element  $r$  in the symmetric function algebra  $A$ .



**Example H146E5**

---

```

> R := Rational();
> P := SFAPower(R);
> m := P!3;
> m;
3
> Parent(m);
Symmetric Algebra over Rational Field, Power sum symmetric functions as basis
> m + P.[3,2];
3 + P.[3,2]

```

---

$A \vdash m$
--------------

Given a symmetric function algebra  $A$  and a symmetric function  $m$  (possibly with a different basis), return the element  $m$  expressed in terms of the basis of  $A$ .

**Example H146E6**

---

A symmetric function can easily be expressed in terms of any of the symmetric function bases, or as a polynomial.

```

> R := Rational();
> S := SFASchur(R);
> H := SFAHomogeneous(R);
> P := SFAPower(R);
> E := SFAElementary(R);
> M := SFAMonomial(R);
>
> m := S.[3,1];
> S!m;
S.[3,1]
> H!m;
H.[3,1] - H.[4]
> P!m;
1/8*P.[1,1,1,1] + 1/4*P.[2,1,1] - 1/8*P.[2,2] - 1/4*P.[4]
> E!m;
E.[2,1,1] - E.[3,1] - E.[2,2] + E.[4]
> M!m;
3*M.[1,1,1,1] + 2*M.[2,1,1] + M.[3,1] + M.[2,2]
>
> PP<[x]> := PolynomialRing(R, 4);
> PP ! m;
x[1]^3*x[2] + x[1]^3*x[3] + x[1]^3*x[4] + x[1]^2*x[2]^2 + 2*x[1]^2*x[2]*x[3] +
2*x[1]^2*x[2]*x[4] + x[1]^2*x[3]^2 + 2*x[1]^2*x[3]*x[4] + x[1]^2*x[4]^2 +
x[1]*x[2]^3 + 2*x[1]*x[2]^2*x[3] + 2*x[1]*x[2]^2*x[4] + 2*x[1]*x[2]*x[3]^2 +
3*x[1]*x[2]*x[3]*x[4] + 2*x[1]*x[2]*x[4]^2 + x[1]*x[3]^3 +
2*x[1]*x[3]^2*x[4] + 2*x[1]*x[3]*x[4]^2 + x[1]*x[4]^3 + x[2]^3*x[3] +

```

$$\begin{aligned}
& x[2]^3 x[4] + x[2]^2 x[3]^2 + 2 x[2]^2 x[3] x[4] + x[2]^2 x[4]^2 + \\
& x[2] x[3]^3 + 2 x[2] x[3]^2 x[4] + 2 x[2] x[3] x[4]^2 + x[2] x[4]^3 + \\
& x[3]^3 x[4] + x[3]^2 x[4]^2 + x[3] x[4]^3
\end{aligned}$$

**Example H146E7**

---

We show that the  $k$ -th homogeneous symmetric function is defined as the sum over all monomial symmetric functions indexed by partitions of weight  $k$ .

```

> R := Rational();
> M := SFA(R : Basis := "Monomial");
> H := SFA(R : Basis := "Homogeneous");
>
> H ! (M.[1]);
H.[1]
> H ! (M.[2] + M.[1,1]);
H.[2]
> H ! (M.[3] + M.[2,1] + M.[1,1,1]);
H.[3]
> H ! (M.[4] + M.[3,1] + M.[2,2] + M.[2,1,1] + M.[1,1,1,1]);
H.[4]
>
> k := 5;
> H ! &+ [ M.P : P in Partitions(k)];
H.[5]
> k := 10;
> H ! &+ [ M.P : P in Partitions(k)];
H.[10]

```

---

## 146.3 Structure Operations

### 146.3.1 Related Structures

The main structure related to an algebra of symmetric functions is its coefficient ring. An algebra of symmetric functions belongs to the MAGMA category `AlgSym`.

BaseRing(L)
-------------

CoefficientRing(L)
--------------------

Return the coefficient ring of an algebra of symmetric functions  $L$ .

Category(L)
-------------

Parent(L)
-----------

PrimeRing(L)
--------------

### 146.3.2 Ring Predicates and Booleans

The usual ring functions returning boolean values are available on algebras of symmetric functions.

IsCommutative(L)	IsUnitary(L)	IsFinite(L)	IsOrdered(L)
IsField(L)	IsEuclideanDomain(L)	IsPID(L)	IsUFD(L)
IsDivisionRing(L)	IsEuclideanRing(L)	IsDomain(L)	
IsPrincipalIdealRing(L)			
L eq M			
L ne M			

Return **true** if  $L$  and  $M$  are equal (respectively, not equal). MAGMA considers two algebras to be equal if they are over the same ring.

### 146.3.3 Predicates on Basis Types

HasSchurBasis(A)
------------------

Returns **true** if  $A$  is an algebra with a Schur basis.

HasHomogeneousBasis(A)
HasElementaryBasis(A)
HasPowerSumBasis(A)
HasMonomialBasis(A)

Returns **true** if  $A$  is an algebra with a homogeneous, elementary, power sum or monomial basis respectively.

## 146.4 Element Operations

### 146.4.1 Parent and Category

The category for elements in algebras of symmetric functions is **AlgSymElt**.

Parent(f)	Category(f)
-----------	-------------

### 146.4.2 Print Styles

By default, elements are printed using a lexicographical ordering on the partitions indexing the basis elements. For partitions of the same weight  $w$  this ordering is the reverse of that in `Partitions(w)`. More generally, basis elements indexed by partitions with smaller entries print first (see 10.5.2). It is possible to rearrange the linear combination of basis elements so that the basis elements indexed by the longest partition print first and also so that the basis elements indexed by the partition with greatest maximal part (first entry) print first.

A ' PrintStyle
----------------

Retrieve or set the style in which elements of the algebra  $A$  will print. The default is the lexicographical ordering described above, "Lex". Other options are "Length" and "MaximalPart".

---

#### Example H146E8

We demonstrate the setting of the `PrintStyle` and the corresponding printing of elements.

```
> M := SFAMonomial(Rationals());
> M'PrintStyle;
Lex
> P := Partitions(3);
> P;
[
  [ 3 ],
  [ 2, 1 ],
  [ 1, 1, 1 ]
]
> f := &+[M.p : p in P];
> f;
M.[1,1,1] + M.[2,1] + M.[3]
> M'PrintStyle := "Length";
> f;
M.[3] + M.[2,1] + M.[1,1,1]
> M'PrintStyle := "MaximalPart";
> f;
M.[1,1,1] + M.[2,1] + M.[3]
```

---

### 146.4.3 Additive Arithmetic Operators

The usual unary and binary ring element operations are available for symmetric functions. It is possible to combine elements of different algebras (so long as the coefficient rings are compatible) in these operations. Where the elements are written with respect to different bases, the result will be written with respect to the basis of the second operand.

$+ a$	$- a$
$a + b$	$a - b$
$a +:= b$	$a -:= b$

#### 146.4.4 Multiplication

Where the elements being multiplied are expressed with respect to different bases, the result will again be expressed with respect to the basis of the second operand. The algorithm used for multiplication is dependent on the bases with respect to which the elements are expressed.

$a * b$
$a *:= b$

The product of the symmetric functions  $a$  and  $b$ .

If  $b$  is expressed with respect to a Schur basis and  $a$  is expressed with respect to a power sum, elementary or monomial basis then the algorithm based on Muirs rule [Mui60] is used. If  $a$  is expressed with respect to a homogeneous basis the algorithm based on the Pieri rule [Mac95] is used and if  $a$  is also expressed with respect to a Schur basis then the method of Schubert polynomials [LS85] is used.

Special algorithms are also used if  $b$  is expressed with respect to a monomial basis and  $a$  is expressed with respect to a homogeneous, elementary or monomial basis.

When both  $a$  and  $b$  have a homogeneous, elementary or power sum basis, the multiplication of basis elements involves the merging of the parts of the partitions. This follows from the definition of the basis elements  $f_\lambda = \prod_i f_{\lambda_i}$  where  $f_\lambda$  is a homogeneous, elementary or power sum basis element.

Otherwise  $a$  is coerced into the parent of  $b$  before the elements are multiplied.

The degree of the result is the sum of the degrees of both the operands.

$a \wedge k$
--------------

---

#### Example H146E9

```
> Q := Rationals();
> s := SFASchur(Q);
> m := SFAMonomial(Q);
> m.[3]*s.[2,1];
s.[2,1,1,1,1] + s.[5,1] - s.[2,2,2] - s.[3,3]
```

To illustrate the merging of the partitions :

```
> E := SFAElementary(Q);
> E.4*E.3*E.1;
E.[4,3,1]
```

This is the definition of  $e_{[4,3,1]}$ .

---

### 146.4.5 Plethysm

Plethysm is also referred to as composition of symmetric functions.

a ~ b

This operator computes the plethysm or composition of the symmetric functions  $a$  and  $b$ . The result is given with respect to the basis of the second operand.

The degree of the result is the product of the degrees of operands which may be very large.

#### Example H146E10

---

```
> Q := Rationals();
> s := SFASchur(Q);
> m := SFAMonomial(Q);
> m.[3]~s.[2,1];
s.[4,1,1,1,1,1] - s.[3,2,1,1,1,1] - s.[5,1,1,1,1] + s.[2,2,2,1,1,1] +
s.[3,3,1,1,1] + s.[6,1,1,1] - s.[2,2,2,2,1] - s.[3,3,2,1] - s.[6,2,1] +
s.[4,4,1] + s.[3,2,2,2] + s.[5,2,2] - s.[4,3,2] + 2*s.[3,3,3] + s.[6,3] -
s.[5,4]
```

---

### 146.4.6 Boolean Operators

IsHomogeneous(s)

Returns **true** if the partitions indexing the basis elements present in the symmetric function  $s$  are all of the same weight. This implies that each term has the same degree so is the same as the polynomial expansion of  $s$  being a homogeneous polynomial.

IsZero(s)

IsOne(s)

IsMinusOne(s)

s eq t

s ne t

Return **true** if the symmetric functions  $s$  and  $t$  are (not) the same.

### 146.4.7 Accessing Elements

#### Coefficient(*s*, *p*)

Given a symmetric function  $s$  return the coefficient of the basis element  $A_p$ , where  $A$  is the parent of  $s$  and  $p$  is a sequence defining a partition. The coefficient may be zero.

#### Support(*s*)

Return two parallel sequences of the partitions indexing the basis elements and the coefficients of those basis elements in the symmetric function  $s$ , which is a linear combination of basis elements.

#### Length(*s*)

Given a symmetric function  $s$ , return the length of  $s$ , i.e., the number of basis elements having non zero coefficients in  $s$ , with respect to the current basis.

#### Example H146E11

---

We pull apart an element and show that we can put it back together again.

```
> H := SFAHomogeneous(Rationals());
> P := Partitions(4);
> f := &+[Random(1, 5)*H.p : p in P];
> f;
H.[1,1,1,1] + 4*H.[2,1,1] + 4*H.[2,2] + 5*H.[3,1] + 5*H.[4]
> s, e := Support(f);
> s, e;
[
  [ 1, 1, 1, 1 ],
  [ 2, 1, 1 ],
  [ 2, 2 ],
  [ 3, 1 ],
  [ 4 ]
]
[ 1, 4, 4, 5, 5 ]
> f eq &+[e[i]*H.s[i] : i in [1 .. Length(f)]];
true
```

---

#### Degree(*s*)

Given a symmetric function  $s$ , return the degree of  $s$ , i.e., the maximal degree of the basis elements having non zero coefficients in  $s$ , which is the maximal weight of the partitions indexing those basis elements.

### 146.4.8 Multivariate Polynomials

A symmetric function may be seen as a polynomial in any number of variables.

P ! s
-------

Return the polynomial expansion of the symmetric function  $s$  in the polynomial ring  $P$ .

#### Example H146E12

---

```
> S := SFASchur(GF(7));
> s := S.[3,1];
```

Now we compute the multivariate polynomial we get by restricting to 5 variables.

```
> G<e1, e2, e3, e4, e5> := PolynomialRing(GF(7), 5);
> p := G!s;
> p;
e1^3*e2 + e1^3*e3 + e1^3*e4 + e1^3*e5 + e1^2*e2^2 + 2*e1^2*e2*e3 + 2*e1^2*e2*e4
+ 2*e1^2*e2*e5 + e1^2*e3^2 + 2*e1^2*e3*e4 + 2*e1^2*e3*e5 + e1^2*e4^2 +
2*e1^2*e4*e5 + e1^2*e5^2 + e1*e2^3 + 2*e1*e2^2*e3 + 2*e1*e2^2*e4 +
2*e1*e2^2*e5 + 2*e1*e2*e3^2 + 3*e1*e2*e3*e4 + 3*e1*e2*e3*e5 + 2*e1*e2*e4^2 +
3*e1*e2*e4*e5 + 2*e1*e2*e5^2 + e1*e3^3 + 2*e1*e3^2*e4 + 2*e1*e3^2*e5 +
2*e1*e3*e4^2 + 3*e1*e3*e4*e5 + 2*e1*e3*e5^2 + e1*e4^3 + 2*e1*e4^2*e5 +
2*e1*e4*e5^2 + e1*e5^3 + e2^3*e3 + e2^3*e4 + e2^3*e5 + e2^2*e3^2 +
2*e2^2*e3*e4 + 2*e2^2*e3*e5 + e2^2*e4^2 + 2*e2^2*e4*e5 + e2^2*e5^2 + e2*e3^3
+ 2*e2*e3^2*e4 + 2*e2*e3^2*e5 + 2*e2*e3*e4^2 + 3*e2*e3*e4*e5 + 2*e2*e3*e5^2
+ e2*e4^3 + 2*e2*e4^2*e5 + 2*e2*e4*e5^2 + e2*e5^3 + e3^3*e4 + e3^3*e5 +
e3^2*e4^2 + 2*e3^2*e4*e5 + e3^2*e5^2 + e3*e4^3 + 2*e3*e4^2*e5 + 2*e3*e4*e5^2
+ e3*e5^3 + e4^3*e5 + e4^2*e5^2 + e4*e5^3
```

To check the polynomial is actually symmetric, we can use the MAGMA intrinsic `IsSymmetric`, which also computes an expansion as a sum of elementary symmetric polynomials.

```
> IsSymmetric(p,G);
true e1^2*e2 + 6*e1*e3 + 6*e2^2 + e4
```

Which is identical to the result of

```
> E := SFAElementary(GF(7));
> E!s;
E.[2,1,1] + 6*E.[2,2] + 6*E.[3,1] + E.[4]
```

#### Example H146E13

---

These conversions may be used to change the alphabet of a symmetric function. For example, if we substitute the variable  $x_i$  by  $x_i + 1$ , the result is again a symmetric function, however we use polynomials to do the evaluation.

```
> S := SFASchur(Rationals());
> R<a, b, c, d, e> := PolynomialRing(Rationals(), 5);
```



```

> p := Polynomial(S.[3,2], R);
> q := Evaluate(p, [a+1, b+1, c+1, d+1, e+1]);
> x, y := IsCoercible(S, q);
> y;
175 + 175*S.[1] + 70*S.[1,1] + 70*S.[2] + 35*S.[2,1] + 7*S.[2,2] + 10*S.[3] +
5*S.[3,1] + S.[3,2]

```

---

### 146.4.9 Frobenius Homomorphism

There is an automorphism which maps the elementary symmetric function to the homogeneous symmetric function.

Frobenius(s)
--------------

Given any symmetric function  $s$ , compute the image of  $s$  under the Frobenius automorphism. The image will have the same parent as  $s$ . When the power sum symmetric functions are involved, it may be necessary to work with a coefficient ring which allows division by an integer.

#### Example H146E14

---

It is known that the Frobenius automorphism on the Schur functions acts just by conjugating the indexing partitions.

```

> S := SFASchur(Integers());
> E := SFAElementary(Integers());
> h := S!E.[3,3,3];
> h;
S.[1,1,1,1,1,1,1,1,1,1] + 2*S.[2,1,1,1,1,1,1,1,1] + S.[3,1,1,1,1,1,1] +
3*S.[2,2,1,1,1,1,1] + 2*S.[3,2,1,1,1,1] + 4*S.[2,2,2,1,1,1] + S.[3,3,1,1,1] +
3*S.[3,2,2,1,1] + 2*S.[2,2,2,2,1] + 2*S.[3,3,2,1] + S.[3,2,2,2] + S.[3,3,3]

```

Now apply the Frobenius automorphism.

```

> f:=Frobenius(h);
> f;
S.[3,3,3] + 2*S.[4,3,2] + S.[4,4,1] + S.[5,2,2] + 3*S.[5,3,1] + 2*S.[5,4] +
2*S.[6,2,1] + 4*S.[6,3] + S.[7,1,1] + 3*S.[7,2] + 2*S.[8,1] + S.[9]

```

To check whether the coefficient of the basis element indexed by a partition in one element is the same as the coefficient of the basis element indexed by the conjugate partition in the other :

```

> p:=Partitions(f);
> for pp in p do
>   if Coefficient(h, ConjugatePartition(pp)) ne Coefficient(f, pp) then
>     print pp;
>   end if;
> end for;

```

---

### 146.4.10 Inner Product

An inner product on  $\Lambda$  is defined by

$$\langle m_\lambda, h_{\lambda'} \rangle = \delta_{\lambda, \lambda'}.$$

This definition ensures that the bases  $m_\lambda$  and  $h_\lambda$  are dual to each other. This is the inner product used by MAGMA.

<code>InnerProduct(a,b)</code>
--------------------------------

Computes the inner product of the symmetric functions  $a$  and  $b$ .

#### Example H146E15

---

The inner product of a elementary symmetric function indexed by a partition and the homogeneous symmetric function indexed by the conjugate partition is 1. (This allows the computation of irreducible representations of the symmetric group.) To check this with MAGMA, we do the following:

```
> E := SFAElementary(Rationals());
> H := SFAHomogeneous(Rationals());
> p:=RandomPartition(45);
> pc:=ConjugatePartition(p);
> InnerProduct(E.p,H.pc);
1
```

As it should be the result is 1.

---

### 146.4.11 Combinatorial Objects

The Schur function is the generating function of standard tableaux. Therefore, it is possible to get the corresponding tableaux.

<code>Tableaux(sf, m)</code>
------------------------------

Given a Schur function  $sf$  over the integers with positive coefficients, return the multiset of the tableaux, with maximal entry  $m$ , for which  $sf$  is the generating function.

### 146.4.12 Symmetric Group Character

A Schur function indexed by a single partition, i.e. a basis element, corresponds to an irreducible character of the symmetric group.

<code>SymmetricCharacter(sf)</code>
-------------------------------------

Given an element  $sf$  of an algebra of symmetric functions return a linear combination of irreducible characters of the symmetric group, whose coefficients are the coefficients of  $sf$  with respect to the Schur function basis.

**Example H146E16**

We look at a result in the representation theory of the symmetric group. There is exactly one irreducible character contained in both the induced character from the identity character of a Young subgroup indexed by the partition  $I$  and the induced character from the alternating character of a Young subgroup indexed by the partition  $J$ , conjugate to  $I$ . As the induced character of the identity character of a Young subgroup corresponds to the homogeneous symmetric function  $h_I$  and the induced character of the alternating character corresponds to the elementary symmetric function  $e_J$ , we can verify this using the following routine:

```
> H := SFAHomogeneous(Rationals());
> E := SFAElementary(Rationals());
> p := Partitions(7);
> for I in p do
>   a := SymmetricCharacter(H.I);
>   J := ConjugatePartition(I);
>   b := SymmetricCharacter(E.J);
>   i := InnerProduct(a,b);
>   if i ne 1 then print i; end if;
> end for;
```

And there should be no output.

**146.4.13 Restrictions**

It is possible to form symmetric functions whose support is a subset of the support of a given symmetric function, subject to some restrictions.

<b>RestrictDegree(a, n)</b>
-----------------------------

**Exact**

BOOLELT

*Default : true*

Return the symmetric function which is a linear combination of those basis elements of the symmetric function  $a$  with degree  $n$ . This is the restriction of  $a$  into the submodule  $\Lambda^n$ . If **Exact** is **false** then the basis elements included will have degree  $\leq n$ . This is the restriction of  $a$  into  $\bigcup_{k \leq n} \Lambda^k$ .

<b>RestrictPartitionLength(a, n)</b>
--------------------------------------

**Exact**

BOOLELT

*Default : true*

Return the symmetric function which is a linear combination of those basis elements of the symmetric function  $a$  whose indexing partitions are of length  $n$ . If **Exact** is **false**, then the indexing partitions will be of length  $\leq n$ .

<b>RestrictParts(a, n)</b>
----------------------------

**Exact**

BOOLELT

*Default : true*

Return the symmetric function which is a linear combination of those basis elements of the symmetric function  $a$  whose indexing partitions have maximal part  $n$ . If **Exact** is **false**, then the indexing partitions will have maximal part  $\leq n$ .

## 146.5 Transition Matrices

One area of interest in the theory of symmetric functions is the study of the change of bases between the five different bases. The matrices of change of base between the  $s_\lambda$ ,  $h_\lambda$ ,  $m_\lambda$  and the  $e_\lambda$  are all integer matrices. Only when changing from one of these four bases to the  $p_\lambda$ , is the matrix over the rationals. For a discussion of their computation and interactions see [Mac95, pages 54–58].

In MAGMA there are routines available to obtain all these matrices. These routines are described below. Some interactions of these matrices are verified in examples.

### 146.5.1 Transition Matrices from Schur Basis

<code>SchurToMonomialMatrix(n)</code>
---------------------------------------

Computes the matrix for the expansion of a Schur function indexed by a partition of weight  $n$  as a sum of monomial symmetric functions. This matrix is also known as the table of *Kostka* numbers. These are the numbers of tableaux of each shape and content. The content of a tableau prescribes its entries, so for example a tableau with content  $[2, 1, 4, 1]$  has two 1's, one 2, four 3's and one 4. The entries of the matrix are non negative integers. The matrix is upper triangular.

---

#### Example H146E17

Compute the base change matrix from the Schur functions to the monomial symmetric functions for degree 5. The entries in this matrix are what are known as the *Kostka* numbers. They count the number of young tableaux on a given shape. Look on the order of the labelling partitions, and check whether the entry 3 in the upper right corner is right by generating the corresponding tableaux.

```
> M := SchurToMonomialMatrix(5);
> M;
[1 1 1 1 1 1 1]
[0 1 1 2 2 3 4]
[0 0 1 1 2 3 5]
[0 0 0 1 1 3 6]
[0 0 0 0 1 2 5]
[0 0 0 0 0 1 4]
[0 0 0 0 0 0 1]
> Parts := Partitions(5);
> Parts;
[
  [ 5 ],
  [ 4, 1 ],
  [ 3, 2 ],
  [ 3, 1, 1 ],
  [ 2, 2, 1 ],
  [ 2, 1, 1, 1 ],
  [ 1, 1, 1, 1, 1 ]
]
```

```
> #TableauxOnShapeWithContent(Parts[2], Parts[6]);
3
> M[2,6];
3
```

#### SchurToHomogeneousMatrix(n)

Computes the matrix for the expansion of a Schur function indexed by a partition of weight  $n$  as a sum of homogeneous symmetric functions. The entries of the matrix are positive and negative integers. The matrix is lower triangular. It is the transpose of the matrix returned by `MonomialToSchurMatrix(n)`.

#### SchurToPowerSumMatrix(n)

Computes the matrix for the expansion of a Schur function indexed by a partition of weight  $n$  as a sum of power sum symmetric functions. The entries of the matrix are rationals.

#### SchurToElementaryMatrix(n)

Computes the matrix for the expansion of a Schur function indexed by a partition of weight  $n$  as a sum of elementary symmetric functions. The entries of the matrix are positive and negative integers. The matrix is upper left triangular.

### Example H146E18

We verify by hand that the action of base change matrix is the same as coercion.

```
> S := SFASchur(Rationals());
> P := SFAPower(Rationals());
> NumberOfPartitions(4);
5
> m := SchurToPowerSumMatrix(4);
> Partitions(4);
[
  [ 4 ],
  [ 3, 1 ],
  [ 2, 2 ],
  [ 2, 1, 1 ],
  [ 1, 1, 1, 1 ]
]
> s := S.[3, 1] + 5*S.[1, 1, 1, 1] - S.[4];
> s;
5*S.[1,1,1,1] + S.[3,1] - S.[4]
> p, c := Support(s);
> c;
[ 5, 1, -1 ]
> p;
[
```

```

      [ 1, 1, 1, 1 ],
      [ 3, 1 ],
      [ 4 ]
]
> cm := Matrix(Rationals(), 1, 5, [-1, 1, 0, 0, 5]);
> cm*m;
[-7/4  4/3  3/8 -5/4 7/24]
> P!s;
7/24*P.[1,1,1,1] - 5/4*P.[2,1,1] + 3/8*P.[2,2] + 4/3*P.[3,1] - 7/4*P.[4]

```

The coefficients of the coerced element are the reverse of the matrix product, consistent with the partition in the coerced element being in reverse order to those in `Partitions(4)`.

---

## 146.5.2 Transition Matrices from Monomial Basis

### `MonomialToSchurMatrix(n)`

Computes the matrix for the expansion of a monomial symmetric function indexed by a partition of weight  $n$  as a sum of Schur symmetric functions. The entries of the matrix are positive and negative integers. The matrix is upper triangular. It is the transpose of the matrix returned by `SchurToHomogeneousMatrix(n)`.

### `MonomialToHomogeneousMatrix(n)`

Computes the matrix for the expansion of a monomial symmetric function indexed by a partition of weight  $n$  as a sum of homogeneous symmetric functions. The entries are positive and negative integers.

### `MonomialToPowerSumMatrix(n)`

Computes the matrix for the expansion of a monomial symmetric function indexed by a partition of weight  $n$  as a sum of power sum symmetric functions. The entries are rationals. The matrix is lower triangular.

### `MonomialToElementaryMatrix(n)`

Computes the matrix for the expansion of a monomial symmetric function indexed by a partition of weight  $n$  as a sum of elementary symmetric functions. The entries of the matrix are positive and negative integers. The matrix is upper left triangular.

### 146.5.3 Transition Matrices from Homogeneous Basis

#### HomogeneousToSchurMatrix(n)

Computes the matrix for the expansion of a homogeneous symmetric function indexed by a partition of weight  $n$  as a sum of Schur symmetric functions. The entries of the matrix are positive integers. The matrix is lower triangular.

#### Example H146E19

---

It is known that the matrix computed by `HomogeneousToSchurMatrix` is the transpose of the matrix computed by `SchurToMonomialMatrix`.

```
> SchurToMonomialMatrix(7) eq Transpose(HomogeneousToSchurMatrix(7));
true
```

---

#### HomogeneousToMonomialMatrix(n)

Computes the matrix  $M$  for the expansion of a homogeneous symmetric function indexed by a partition of weight  $n$  as a sum of monomial symmetric functions. The entries of the matrix are positive integers. The matrix has no zero entries.

The coefficient  $M_{\mu,\lambda}$  in the expansion  $h_\lambda = \sum_\mu M_{\mu,\lambda} m_\mu$  is the number of non negative integer matrices with row sums  $\lambda_i$  and column sums  $\mu_j$ , see [Mac95, page 57].

#### Example H146E20

---

The matrix converting from homogeneous basis to monomial basis is symmetric.

```
> IsSymmetric(HomogeneousToMonomialMatrix(7));
true
```

---

#### HomogeneousToPowerSumMatrix(n)

Computes the matrix for the expansion of a homogeneous symmetric function indexed by a partition of weight  $n$  as a sum of power sum symmetric functions. The entries of the matrix are positive rationals. The matrix is upper triangular.

#### HomogeneousToElementaryMatrix(n)

Computes the matrix for the expansion of a homogeneous symmetric function indexed by a partition of weight  $n$  as a sum of elementary symmetric functions. The entries of the matrix are integers. The matrix is upper triangular.

**Example H146E21**

It is known that the matrix compute by `HomogeneousToElementaryMatrix` is the same as the matrix computed by `ElementaryToHomogeneousMatrix`.

```
> HomogeneousToElementaryMatrix(7) eq ElementaryToHomogeneousMatrix(7);
true
```

**146.5.4 Transition Matrices from Power Sum Basis****PowerSumToSchurMatrix(n)**

Computes the matrix for the expansion of a power sum symmetric function indexed by a partition of weight  $n$  as a sum of Schur symmetric functions. The entries of the matrix are positive and negative integers. This matrix is the character table of the symmetric group.

**Example H146E22**

The matrix returned by `PowerSumToSchurMatrix` is compared to the character table of the appropriate symmetric group.

```
> PowerSumToSchurMatrix(5);
[ 1 -1  0  1  0 -1  1]
[ 1  0 -1  0  1  0 -1]
[ 1 -1  1  0 -1  1 -1]
[ 1  1 -1  0 -1  1  1]
[ 1  0  1 -2  1  0  1]
[ 1  2  1  0 -1 -2 -1]
[ 1  4  5  6  5  4  1]
> CharacterTable(Sym(5));
```

Character Table

-----

```
-----
Class |  1  2  3  4  5  6  7
Size  |  1 10 15 20 30 24 20
Order |  1  2  2  3  4  5  6
-----
p  =  2  1  1  1  4  3  6  4
p  =  3  1  2  3  1  5  6  2
p  =  5  1  2  3  4  5  1  7
-----
X.1  +  1  1  1  1  1  1  1
X.2  +  1 -1  1  1 -1  1 -1
```



```

X.3  +  4  2  0  1  0 -1 -1
X.4  +  4 -2  0  1  0 -1  1
X.5  +  5  1  1 -1 -1  0  1
X.6  +  5 -1  1 -1  1  0 -1
X.7  +  6  0 -2  0  0  1  0

```

In the character table the first row is the unity character, which corresponds to the first column of the transition matrix. The second row of the character table is the alternating character which corresponds to the last column of the transition matrix. The first column of the character table contains the dimensions of the irreducible characters, this is the last row of the transition matrix.

---

#### PowerSumToMonomialMatrix(n)

Computes the matrix for the expansion of a power sum symmetric function indexed by a partition of weight  $n$  as a sum of monomial symmetric functions. The entries of the matrix are positive integers. The matrix is lower triangular.

#### PowerSumToHomogeneousMatrix(n)

Computes the matrix for the expansion of a power sum symmetric function indexed by a partition of weight  $n$  as a sum of homogeneous symmetric functions. The entries of the matrix are integers. The matrix is upper triangular.

#### PowerSumToElementaryMatrix(n)

Computes the matrix for the expansion of a power sum symmetric function indexed by a partition of weight  $n$  as a sum of elementary symmetric functions. The entries of the matrix are integers. The matrix is upper triangular.

### 146.5.5 Transition Matrices from Elementary Basis

#### ElementaryToSchurMatrix(n)

Computes the matrix for the expansion of an elementary symmetric function indexed by a partition of weight  $n$  as a sum of Schur symmetric functions. The entries of the matrix are positive integers.

#### ElementaryToMonomialMatrix(n)

Computes the matrix  $M$  for the expansion of an elementary symmetric function indexed by a partition of weight  $n$  as a sum of monomial symmetric functions. The entries of the matrix are positive integers.

The coefficient  $M_{\mu,\lambda}$  in the expansion  $e_\lambda = \sum_\mu M_{\mu,\lambda} m_\mu$  is the number of 0-1 integer matrices with row sum  $\lambda_i$  and column sum  $\mu_j$ , see [Mac95, page 57].

**Example H146E23**

---

The matrix converting from elementary basis to monomial basis is symmetric.

```
> IsSymmetric(ElementaryToMonomialMatrix(7));  
true
```

---

**ElementaryToHomogeneousMatrix( $n$ )**

Computes the matrix for the expansion of a elementary symmetric function indexed by a partition of weight  $n$  as a sum of homogeneous symmetric functions.

**ElementaryToPowerSumMatrix( $n$ )**

Computes the matrix for the expansion of a elementary symmetric function indexed by a partition of weight  $n$  as a sum of power sum symmetric functions.

## 146.6 Bibliography

- [LS85] Alain Lascoux and Marcel-Paul Schützenberger. Schubert polynomials and the Littlewood-Richardson rule. *Lett. Math. Phys.*, 10(2-3):111–124, 1985.
- [Mac95] I. G. Macdonald. *Symmetric functions and Hall polynomials*. The Clarendon Press Oxford University Press, New York, second edition, 1995. With contributions by A. Zelevinsky, Oxford Science Publications.
- [Mui60] Thomas Muir. *A treatise on the theory of determinants*. Dover Publications Inc., New York, 1960.

# 147 INCIDENCE STRUCTURES AND DESIGNS

<b>147.1 Introduction . . . . .</b>	<b>4873</b>	<i>147.4.2 The Witt Designs . . . . .</i>	<i>4884</i>
<b>147.2 Construction of Incidence Structures and Designs . . .</b>	<b>4874</b>	WittDesign(n)	4884
IncidenceStructure< >	4874	<i>147.4.3 Difference Sets and their Develop- ment . . . . .</i>	<i>4884</i>
IncidenceStructure< >	4874	DifferenceSet(p, t)	4884
NearLinearSpace< >	4874	SingerDifferenceSet(n, q)	4884
NearLinearSpace< >	4874	IsDifferenceSet(B)	4885
LinearSpace< >	4875	Development(B)	4885
LinearSpace< >	4875	Development(T)	4885
Design< >	4876	<b>147.5 Elementary Invariants of an In- cidence Structure . . . . .</b>	<b>4886</b>
Design< >	4876	NumberOfPoints(D)	4886
<b>147.3 The Point-Set and Block-Set of an Incidence Structure . . .</b>	<b>4878</b>	#	4886
<i>147.3.1 Introduction . . . . .</i>	<i>4878</i>	Points(D)	4886
<i>147.3.2 Creating Point-Sets and Block-Sets</i>	<i>4879</i>	Support(D)	4886
PointSet(D)	4879	PointDegrees(D)	4886
BlockSet(D)	4879	NumberOfBlocks(D)	4886
<i>147.3.3 Creating Points and Blocks . . . . .</i>	<i>4879</i>	#	4886
Point(D, i)	4879	Blocks(D)	4886
.	4879	BlockDegrees(D)	4887
Representative(P)	4879	BlockSizes(D)	4887
Rep(P)	4879	Covalence(D, S)	4887
Random(P)	4879	IncidenceMatrix(D)	4887
!	4879	pRank(D, p)	4887
Block(D, i)	4879	<b>147.6 Elementary Invariants of a De- sign . . . . .</b>	<b>4887</b>
.	4879	Parameters(D)	4887
Representative(B)	4880	ReplicationNumber(D)	4887
Rep(B)	4880	BlockDegree(D)	4887
Random(B)	4880	BlockSize(D)	4887
!	4880	Covalence(D, s)	4887
Representative(b)	4880	Order(D)	4887
Rep(b)	4880	IntersectionNumber(D, i, j)	4887
Random(b)	4880	PascalTriangle(D)	4888
<b>147.4 General Design Constructions</b>	<b>4881</b>	<b>147.7 Operations on Points and Blocks . . . . .</b>	<b>4889</b>
<i>147.4.1 The Construction of Related Struc- tures . . . . .</i>	<i>4881</i>	in	4889
Complement(D)	4881	notin	4889
Dual(D)	4881	subset	4889
Contraction(D, p)	4881	notsubset	4889
Contraction(D, b)	4881	PointDegree(D, p)	4889
Residual(D, b)	4882	BlockDegree(D, B)	4889
Residual(D, p)	4882	BlockSize(D, B)	4889
Simplify(D)	4882	#	4889
Sum(Q)	4882	Set(B)	4890
Union(D, E)	4882	Support(B)	4890
Restriction(D, S)	4882	IsBlock(D, S)	4890
		Line(D, p, q)	4890

Block(D, p, q)	4890	eq	4897
ConnectionNumber(D, p, B)	4890	ne	4897
<b>147.8 Elementary Properties of Incidence Structures and Designs</b>	<b>4891</b>	IsIsomorphic(D, E: -)	4898
IsSimple(D)	4891	<b>147.12 The Automorphism Group of an Incidence Structure . . .</b>	<b>4898</b>
IsTrivial(D)	4891	147.12.1 Construction of Automorphism Groups . . . . .	4898
IsSelfDual(D)	4891	AutomorphismGroup(D)	4898
IsUniform(D)	4891	AutomorphismSubgroup(D)	4899
IsNearLinearSpace(D)	4892	AutomorphismGroupStabilizer(D, k)	4899
IsLinearSpace(D)	4892	PointGroup(D)	4899
IsDesign(D, t: -)	4892	BlockGroup(D)	4899
IsBalanced(D, t: -)	4892	Aut(D)	4900
IsComplete(D)	4892	147.12.2 Action of Automorphisms . . .	4901
IsSymmetric(D)	4892	Image(g, Y, y)	4901
IsSteiner(D, t)	4892	Orbit(G, Y, y)	4901
IsPointRegular(D)	4892	Orbits(G, Y)	4901
IsLineRegular(D)	4892	Stabilizer(G, Y, y)	4901
<b>147.9 Resolutions, Parallelisms and Parallel Classes . . . . .</b>	<b>4893</b>	Action(G, Y)	4901
HasResolution(D)	4893	ActionImage(G, Y)	4901
HasResolution(D, $\lambda$ )	4893	ActionKernel(G, Y)	4902
AllResolutions(D)	4893	IsPointTransitive(D)	4902
AllResolutions(D, $\lambda$ )	4893	IsBlockTransitive(D)	4902
IsResolution(D, P)	4893	<b>147.13 Incidence Structures, Graphs and Codes . . . . .</b>	<b>4903</b>
HasParallelism(D: -)	4893	IncidenceStructure(G)	4903
AllParallelisms(D)	4894	PointGraph(D)	4903
IsParallelism(D, P)	4894	BlockGraph(D)	4903
HasParallelClass(D)	4894	IncidenceGraph(D)	4903
IsParallelClass(D, B, C)	4894	LinearCode(D, K)	4903
AllParallelClasses(D)	4894	<b>147.14 Automorphisms of Matrices</b>	<b>4904</b>
<b>147.10 Conversion Functions . . .</b>	<b>4896</b>	~	4904
IncidenceStructure(I)	4896	AutomorphismGroup(M)	4904
NearLinearSpace(I)	4896	IsIsomorphic(M, N)	4904
LinearSpace(I)	4897	<b>147.15 Bibliography . . . . .</b>	<b>4905</b>
Design(I, t)	4897		
<b>147.11 Identity and Isomorphism .</b>	<b>4897</b>		

# Chapter 147

## INCIDENCE STRUCTURES AND DESIGNS

### 147.1 Introduction

Since there is some variation between authors of the terminology employed in design theory, we begin with some definitions. An *incidence structure* is a triple  $D = (P, B, I)$ , where:

- (a)  $P$  is a set, the elements of which are called *points*;
- (b)  $B$  is a set, the elements of which are called *blocks*;
- (c)  $I$  is an incidence relation between  $P$  and  $B$ , so that  $I \subset P \times B$ . The elements of  $I$  are called *flags*.

Usually, blocks will be subsets of  $P$ , so that instead of writing  $(p, b) \in I$ , we write  $p \in b$ . In general, repeated blocks are allowed so that different blocks may correspond to the same subset of  $P$ . If  $D$  has no repeated blocks, then we say that  $D$  is *simple*.

An incidence structure  $D$  is said to be *uniform* with *blocksize*  $k$  if  $D$  has at least one block and all blocks contain exactly  $k$  points. A uniform incidence structure is called *trivial* if each  $k$ -subset of the point set appears as a block (at least once).

Let  $t \geq 0$  be an integer. Then an incidence structure  $D$  is said to be  *$t$ -balanced* if there exists an integer  $\lambda \geq 1$  such that each  $t$ -subset of the point set is contained in exactly  $\lambda$  blocks of  $D$ .

A *near-linear space* is an incidence structure in which every block contains at least two points and any two points lie in at most one block. A *linear space* is a near-linear space in which any two points lie in *exactly* one block. It is usual, when discussing near-linear spaces, to use the term *line* in place of the term *block*.

Let  $v, k, t$  and  $\lambda$  be integers with  $v \geq k \geq t \geq 0$  and  $\lambda \geq 1$ . A  *$t$ -design* with  $v$  points and *blocksize*  $k$  is an incidence structure  $D = (P, B, I)$  where:

- (a) The cardinality of  $P$  is  $v$ ;
- (b)  $D$  is uniform with blocksize  $k$ ;
- (c)  $D$  is simple;
- (d) For each  $t$ -subset  $T$  of  $P$  there are exactly  $\lambda$  blocks of  $B$  incident with all the points of  $T$  (so  $D$  is  $t$ -balanced).

Such a design is usually referred to as a  $t$ -( $v, k, \lambda$ ) design. The parameter  $\lambda$  is called the *index* of the design. If  $b$  denotes the cardinality of  $B$ , a  $t$ -design with  $v = b$  and  $t \geq 2$  is called a *symmetric design*. A  $t$ -design with  $\lambda = 1$  is called a *Steiner design*. A design which is trivial is also called a *complete design*. Note that a design  $D$  must contain at least one block (i.e.  $b > 0$ ).

The category names for the different families of incidence structures are as follows:

- Incidence structure : `Inc`
- Near-linear space : `IncNsp`
- Linear space : `IncLsp`
- $t$ -design : `Dsgn`

## 147.2 Construction of Incidence Structures and Designs

<code>IncidenceStructure&lt; v   X &gt;</code>
<code>IncidenceStructure&lt; P   X &gt;</code>

Construct the incidence structure  $D$  having point set  $P = \{@p_1, p_2, \dots, p_v@\}$  (where  $p_i = i$  for each  $i$  if the first form of the constructor is used), and block set  $B = \{B_1, B_2, \dots, B_b\}$  given by  $X$ . The value of  $X$  must be either:

- A list of subsets of the set  $P$ .
- A sequence, set or indexed set of subsets of  $P$ .
- A list of blocks of an existing incidence structure.
- A sequence, set or indexed set of blocks of an existing incidence structure.
- A combination of the above.
- A  $v \times b$   $(0,1)$ -matrix  $A$ , where  $A$  may be defined over any coefficient ring. The matrix  $A$  will be interpreted as the incidence matrix for the incidence structure  $D$ .
- A set of codewords of a linear code with length  $v$ . The block set of  $D$  is taken to be the set of supports of the codewords.

The incidence structure  $D$  produced by this constructor will have  $P$  as its point set and  $B$  as its set of blocks. The function returns three values:

- The incidence structure  $D$ ;
- The point-set  $P$  for  $D$ ; and
- The block-set  $B$  for  $D$ .

<code>NearLinearSpace&lt; v   X : parameters &gt;</code>
<code>NearLinearSpace&lt; P   X : parameters &gt;</code>

Check

BOOLELT

Default : true

Construct the near-linear space  $S$  on the set of points  $P = \{@p_1, p_2, \dots, p_v@\}$  (where  $p_i = i$  for each  $i$  if the first form of the constructor is used), with lines  $L = \{L_1, L_2, \dots, L_b\}$  given by  $X$ . The value of  $X$  must be either:

- A list of subsets of the set  $P$ .
- A sequence, set or indexed set of subsets of  $P$ .
- A list of blocks of an existing incidence structure.

- (d) A sequence, set or indexed set of blocks of an existing incidence structure.
- (e) A combination of the above.
- (f) A  $v \times b$   $(0,1)$ -matrix  $A$ , where  $A$  may be over any coefficient ring. The matrix  $A$  will be interpreted as the incidence matrix for the near-linear space  $S$ .
- (g) A set of codewords of a linear code with length  $v$ . The line set of  $S$  is taken to be the set of supports of the codewords.

The set of lines  $L$  defined by  $X$  must satisfy two properties:

- (a) Each line of  $L$  must contain at least two points;
- (b) Any two points of  $P$  may lie on at most one line.

The optional boolean argument **Check** indicates whether or not to check that these two properties are satisfied.

The near-linear space  $S$  produced by this constructor will have  $P$  as its point set and  $L$  as its set of lines. The function returns three values:

- (i) The near-linear space  $S$ ;
- (ii) The point-set  $P$  for  $S$ ; and
- (iii) The line-set  $L$  for  $S$ .

<b>LinearSpace</b> < $v$   $X$ : <i>parameters</i> >
<b>LinearSpace</b> < $P$   $X$ : <i>parameters</i> >

**Check**

BOOLELT

*Default : true*

Construct the linear space  $S$  on the set of points  $P = \{@p_1, p_2, \dots, p_v@\}$  (where  $p_i = i$  for each  $i$  if the first form of the constructor is used), with lines  $L = \{L_1, L_2, \dots, L_b\}$  given by  $X$ . The value of  $X$  must be either:

- (a) A list of subsets of the set  $P$ .
- (b) A sequence, set or indexed set of subsets of  $P$ .
- (c) A list of blocks of an existing incidence structure.
- (d) A sequence, set or indexed set of blocks of an existing incidence structure.
- (e) A combination of the above.
- (f) A  $v \times b$   $(0,1)$ -matrix  $A$ , where  $A$  may be defined over any coefficient ring. The matrix  $A$  will be interpreted as the incidence matrix for the linear space  $S$ .
- (g) A set of codewords of a linear code with length  $v$ . The line set of  $S$  is taken to be the set of supports of the codewords.

The set of lines  $L$  defined by  $X$  must satisfy two properties:

- (a) Each line of  $L$  must contain at least two points;
- (b) Any two points of  $P$  must lie on precisely one line.

The optional boolean argument **Check** indicates whether or not to check that these two properties are satisfied.

The linear space  $S$  produced by this constructor will have  $P$  as its point set and  $L$  as its set of lines. The function returns three values:

- (i) The linear space  $S$ ;
- (ii) The point-set  $P$  for  $S$ ; and
- (iii) The line-set  $L$  for  $S$ .

Design< $t, v \mid X : parameters$ >
--------------------------------------

Design< $t, P \mid X : parameters$ >
--------------------------------------

**Check**

BOOLELT

*Default : true***A1**

MONSTG

*Default : "NoOrbits"*

Construct the  $t$ -design  $D$  on the set of points  $P = \{@p_1, p_2, \dots, p_v@\}$  (where  $p_i = i$  for each  $i$  if the first form of the constructor is used), with blocks  $B = \{B_1, B_2, \dots, B_b\}$  given by  $X$ . The value of  $X$  must be either:

- (a) A list of subsets of the set  $P$ .
- (b) A sequence, set or indexed set of subsets of  $P$ .
- (c) A list of blocks of an existing incidence structure.
- (d) A sequence, set or indexed set of blocks of an existing incidence structure.
- (e) A combination of the above.
- (f) A  $v \times b$   $(0, 1)$ -matrix  $A$ , where  $A$  may be over any coefficient ring. The matrix  $A$  will be interpreted as the incidence matrix for the  $t$ -( $v, k, \lambda$ ) design  $D$ .
- (g) A set of codewords of a linear code with length  $v$ . The block set of  $D$  is taken to be the set of supports of the codewords.

The set of blocks  $B$  defined by  $X$  must satisfy three properties:

- (a) Each block must contain the same number,  $k$  say, of points.
- (b) Every  $t$ -subset of  $P$  must lie in the same number,  $\lambda$  say (where  $\lambda > 0$ ), of blocks.
- (c) There must be no repeated blocks.

The optional boolean argument **Check** indicates whether or not to check that these three properties are satisfied. The optional parameter **A1** can be used to specify the algorithm used for balance testing, see the introduction to Section 147.8 for a full description of its usage.

The  $t$ -( $v, k, \lambda$ ) design  $D$  produced by this constructor will have  $P$  as its point set and  $B$  as its set of blocks. The function returns three values:

- (i) The design  $D$ ;
- (ii) The point-set  $P$  for  $D$ ; and
- (iii) The block-set  $B$  for  $D$ .



**Example H147E1**

---

The Fano plane, considered as a  $2-(7, 3, 1)$  design, may be constructed by the following statement:

```
> F := Design< 2, 7 | {1,2,3}, {1,4,5}, {1,6,7}, {2,4,7},
>                               {2,5,6}, {3,5,7}, {3,4,6} >;
> F: Maximal;
2-(7, 3, 1) Design with 7 blocks
Points: {@ 1, 2, 3, 4, 5, 6, 7 @}
Blocks:
  {1, 2, 3},
  {1, 4, 5},
  {1, 6, 7},
  {2, 4, 7},
  {2, 5, 6},
  {3, 5, 7},
  {3, 4, 6}
```

General incidence structures are allowed repeated blocks, as in the following:

```
> S := IncidenceStructure< {@ 4, 5, 7, 9 @} | [{4, 5, 7}, {7, 9},
>                               {5, 7, 9} , {7, 9}] >;
> S: Maximal;
Incidence Structure on 4 points with 4 blocks
Points: {@ 4, 5, 7, 9 @}
Blocks:
  {4, 5, 7},
  {7, 9},
  {5, 7, 9},
  {7, 9}
```

We now construct a linear space by giving its incidence matrix:

```
> R := RMatrixSpace(Integers(), 5, 6);
> I := R![ 1, 0, 1, 1, 0, 0,
>          1, 0, 0, 0, 1, 1,
>          1, 1, 0, 0, 0, 0,
>          0, 1, 1, 0, 1, 0,
>          0, 1, 0, 1, 0, 1];
> L := LinearSpace< 5 | I >;
> L: Maximal;
Linear Space on 5 points with 6 lines
Points: {@ 1, 2, 3, 4, 5 @}
Lines:
  {1, 2, 3},
  {3, 4, 5},
  {1, 4},
  {1, 5},
  {2, 4},
```

$\{2, 5\}$

Finally, we use the minimum weight codewords of the unextended binary Golay code to construct a 4-(23, 7, 1) design. Since we know that this is indeed a 4-design, we set the checking parameter to `false`.

```
> C := GolayCode(GF(2), false);
> C;
[23, 12, 7] Unextended Golay Code over GF(2)
Generator matrix:
[1 0 0 0 0 0 0 0 0 0 0 0 0 1 0 1 0 1 1 1 0 0 0 1]
[0 1 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 0 0 1 0 0 1]
[0 0 1 0 0 0 0 0 0 0 0 0 0 1 1 0 1 0 0 1 0 1 0 1]
[0 0 0 1 0 0 0 0 0 0 0 0 0 1 1 0 0 0 1 1 1 0 1 1]
[0 0 0 0 1 0 0 0 0 0 0 0 0 1 1 0 0 1 1 0 1 1 0 0]
[0 0 0 0 0 1 0 0 0 0 0 0 0 1 1 0 0 1 1 0 1 1 0]
[0 0 0 0 0 0 1 0 0 0 0 0 0 1 1 0 0 1 1 0 1 1]
[0 0 0 0 0 0 0 1 0 0 0 0 1 0 1 1 0 1 1 1 1 0 0]
[0 0 0 0 0 0 0 0 1 0 0 0 0 1 0 1 1 0 1 1 1 1 0]
[0 0 0 0 0 0 0 0 0 1 0 0 0 0 1 0 1 1 0 1 1 1 1]
[0 0 0 0 0 0 0 0 0 0 1 0 1 0 1 1 1 0 0 0 1 1 0]
[0 0 0 0 0 0 0 0 0 0 0 1 0 1 0 1 1 1 0 0 0 1 1]
> minwt := MinimumWeight(C);
> minwt;
7
> wds := Words(C, minwt);
> D := Design< 4, Length(C) | wds : Check := false >;
> D;
4-(23, 7, 1) Design with 253 blocks
```

---

## 147.3 The Point-Set and Block-Set of an Incidence Structure

### 147.3.1 Introduction

An incidence structure created by Magma consists of three objects: the *point-set*  $P$ , the *block-set*  $B$  and the incidence structure  $D$  itself.

Although called the point-set and block-set,  $P$  and  $B$  are not actual MAGMA sets. They simply act as the parent structures for the points and blocks (respectively) of the incidence structure  $D$ , enabling easy creation of these objects via the `!` and `.` operators.

The point-set  $P$  belongs to the MAGMA category `IncPtSet`, and the block-set  $B$  to the category `IncBlkSet`.

In this section, the functions used to create point-sets, block-sets and the points and blocks themselves are described.

### 147.3.2 Creating Point-Sets and Block-Sets

As mentioned above, the point-set and block-set are returned as the second and third arguments of any function which creates an incidence structure. They can also be created via the following two functions.

**PointSet( $D$ )**

Given an incidence structure  $D$ , return the point-set  $P$  of  $D$ .

**BlockSet( $D$ )**

Given an incidence structure  $D$ , return the block-set  $B$  of  $D$ .

### 147.3.3 Creating Points and Blocks

For efficiency and clarity, the points and blocks of an incidence structure are given special types in MAGMA. The category names for points and blocks are `IncPt` and `IncBlk`, respectively. They can be created in the following ways.

**Point( $D, i$ )**

The  $i$ -th point of the incidence structure  $D$ .

**$P . i$**

Given the point-set  $P$  of an incidence structure  $D$  and an integer  $i$ , return the  $i$ -th point of  $D$ .

**Representative( $P$ )**

**Rep( $P$ )**

Given the point-set  $P$  of an incidence structure  $D$ , return a representative point of  $D$ .

**Random( $P$ )**

Given the point-set  $P$  of an incidence structure  $D$ , return a random point of  $D$ .

**$P ! x$**

Given the point-set  $P$  of an incidence structure  $D$ , return the point of  $D$  corresponding to the element  $x$  of the indexed set used to create  $D$ .

**Block( $D, i$ )**

The  $i$ -th block of the incidence structure  $D$ .

**$B . i$**

Given the block-set  $B$  of an incidence structure  $D$  and an integer  $i$ , return the  $i$ -th block of  $D$ .

Representative(B)
-------------------

Rep(B)
--------

Given the block-set  $B$  of an incidence structure  $D$ , return a representative block of  $D$ .

Random(B)
-----------

Given the block-set  $B$  of an incidence structure  $D$ , return a random block of  $D$ .

$B \uparrow S$
----------------

Given the block-set  $B$  of an incidence structure  $D$ , and a set  $S$ , tries to coerce  $S$  into  $B$ .

Representative(b)
-------------------

Rep(b)
--------

Given a block  $b$  of an incidence structure  $D$ , return a representative point of  $D$  which is incident with  $b$ .

Random(b)
-----------

Given a block  $b$  of an incidence structure  $D$ , return a random point incident with  $b$ .

### Example H147E2

---

The following example shows how points and blocks of an incidence structure can be created.

```
> V := {@ 2, 4, 6, 8, 10 @};
> D, P, B := IncidenceStructure< V | {2, 4, 6}, {2, 8, 10}, {4, 6, 8} >;
> D;
Incidence Structure on 5 points with 3 blocks
> P;
Point-set of Incidence Structure on 5 points with 3 blocks
> B;
Block-set of Incidence Structure on 5 points with 3 blocks
> B.2;
{2, 8, 10}
> P.4;
8
> P!4;
4
> P.5 eq Point(D, 5);
true
> b := Random(B);
> b;
{2, 4, 6}
> Parent(b);
Block-set of Incidence Structure on 5 points with 3 blocks
> p := Rep(b);
> p;
```

```

2
> Parent(p);
Point-set of Incidence Structure on 5 points with 3 blocks
> B!{2, 8, 10};
{2, 8, 10}

```

---

## 147.4 General Design Constructions

Each of these functions returns three values:

- (i) The incidence structure  $D$ ;
- (ii) The point-set  $P$  of  $D$ ;
- (iii) The block-set  $B$  of  $D$ .

### 147.4.1 The Construction of Related Structures

All operations defined for incidence structures apply also to near-linear spaces, linear spaces and designs.

Complement(D)

The complement of the incidence structure  $D$ .

Dual(D)

The dual of the incidence structure  $D$ .

Contraction(D, p)

Given an incidence structure  $D = (P, B)$ , and a point  $p \in P$ , form the incidence structure

$$E = (P - \{p\}, \{b - \{p\} : b \in B | p \in b\}).$$

Thus,  $E$  is constructed from  $D$  by deleting  $p$  and retaining only those blocks incident with it.

Contraction(D, b)

Given an incidence structure  $D = (P, B)$ , and a block  $b \in B$ , form the incidence structure

$$E = (b, \{b \cap c : c \in B | c \neq b\}).$$

Thus,  $E$  has point set  $b$  and its blocks are the non-empty intersections of  $b$  with the blocks of  $D$  other than  $b$  itself.

**Residual(*D*, *b*)**

Given an incidence structure  $D = (P, B)$ , and a block  $b \in B$ , form the incidence structure

$$E = (P - b, B - \{b\}).$$

Thus,  $E$  has point set  $P - b$  and its blocks are the non-empty intersections of  $P - b$  with the blocks of  $D$ .

**Residual(*D*, *p*)**

Given an incidence structure  $D = (P, B)$ , and a point  $p \in P$ , form the incidence structure

$$E = (P - \{p\}, \{x : x \in B \mid p \notin x\}).$$

Thus,  $E$  has point set  $P - \{p\}$  and its blocks are the blocks of  $D$  which do not contain  $p$ .

**Simplify(*D*)**

Simplify the incidence structure  $D$ ; i.e., remove repeated blocks from  $D$ .

**Sum(*Q*)**

Given a sequence  $Q = [D_1, \dots, D_l]$  of incidence structures, each of which is defined over the same set  $P$  of points, form the incidence structure obtained by taking the union of the block sets of  $D_1, \dots, D_l$ . Thus, if  $D_i = (P, B_i)$  then  $D = (P, B_1 \cup \dots \cup B_l)$ .

**Union(*D*, *E*)**

The union of incidence structures  $D$  and  $E$ . That is, if  $D = (P, B)$  and  $E = (Q, C)$ , then return  $U = (P \cup Q, B \cup C)$ . The point sets  $P$  and  $Q$  must be disjoint.

**Restriction(*D*, *S*)**

The restriction of the (near-)linear space  $D$  to the set of points  $S$ .

**Example H147E3**

We illustrate some of the above functions with an example.

```
> K := Design< 3, 8 | {1,3,7,8}, {1,2,4,8}, {2,3,5,8}, {3,4,6,8}, {4,5,7,8},
> {1,5,6,8}, {2,6,7,8}, {1,2,3,6}, {1,2,5,7}, {1,3,4,5}, {1,4,6,7}, {2,3,4,7},
> {2,4,5,6}, {3,5,6,7} >;
> CK := Contraction(K, Point(K, 8));
> RK := Residual(K, Block(K, 1));
> K: Maximal;
3-(8, 4, 1) Design with 14 blocks
Points: {@ 1, 2, 3, 4, 5, 6, 7, 8 @}
Blocks:
  {1, 3, 7, 8},
  {1, 2, 4, 8},
```

```

    {2, 3, 5, 8},
    {3, 4, 6, 8},
    {4, 5, 7, 8},
    {1, 5, 6, 8},
    {2, 6, 7, 8},
    {1, 2, 3, 6},
    {1, 2, 5, 7},
    {1, 3, 4, 5},
    {1, 4, 6, 7},
    {2, 3, 4, 7},
    {2, 4, 5, 6},
    {3, 5, 6, 7}
> CK: Maximal;
2-(7, 3, 1) Design with 7 blocks
Points: {0 1, 2, 3, 4, 5, 6, 7 0}
Blocks:
    {1, 3, 7},
    {1, 2, 4},
    {2, 3, 5},
    {3, 4, 6},
    {4, 5, 7},
    {1, 5, 6},
    {2, 6, 7}
> RK: Maximal;
Incidence Structure on 4 points with 13 blocks
Points: {0 2, 4, 5, 6 0}
Blocks:
    {2, 4},
    {2, 5},
    {4, 6},
    {4, 5},
    {5, 6},
    {2, 6},
    {2, 6},
    {2, 5},
    {4, 5},
    {4, 6},
    {2, 4},
    {2, 4, 5, 6},
    {5, 6}
> RKS := Simplify(RK);
> RKS: Maximal;
Incidence Structure on 4 points with 7 blocks
Points: {0 2, 4, 5, 6 0}
Blocks:
    {2, 4},
    {2, 5},
    {4, 6},

```

$\{4, 5\},$   
 $\{5, 6\},$   
 $\{2, 6\},$   
 $\{2, 4, 5, 6\}$

---

### 147.4.2 The Witt Designs

The 5-(12, 6, 1) and 5-(24, 8, 1) designs constructed by Witt, also known as the small and large Mathieu designs, respectively, can be constructed in Magma with the following function.

WittDesign(n)

The Witt 5-design on  $n$  points, where  $n = 12$  or  $24$ .

#### Example H147E4

---

We construct the Witt 5-(24, 8, 1) design and take its contraction at a point. This contraction is in fact isomorphic to the design constructed above from the unextended binary Golay code.

```

> D, P, B := WittDesign(24);
> D;
5-(24, 8, 1) Design with 759 blocks
> p := P.1;
> Cp := Contraction(D, p);
> Cp;
4-(23, 7, 1) Design with 253 blocks

```

---

### 147.4.3 Difference Sets and their Development

Let  $G$  be a group of order  $v$  and let  $k$  and  $\lambda$  be positive integers such that  $1 < k < v$ . A  $(v, k, \lambda)$  *difference set* for  $G$  is a set  $D$  of  $k$  group elements such that the set

$$\{gh^{-1} : g, h \in D | g \neq h\}$$

contains every non-identity element of  $G$  exactly  $\lambda$  times.

DifferenceSet(p, t)

The difference set of type given by  $t$  (which must be one of "Q", "H6", "T", "B", "B0", "0", "00", or "W4") corresponding to the prime  $p$ . The types have the same interpretation as given by Marshall Hall in [Hal86], pp. 141–142.

SingerDifferenceSet(n, q)

The Singer difference set corresponding to a hyperplane of  $\text{PG}(n, q)$ .



**IsDifferenceSet(B)**

Returns **true** iff  $B$  is a difference set over an integer residue class ring or a finite group (with an iterator). If **true**, the value of the parameter  $\lambda$  (i.e., the number of times each non-identity group/ring element appears as a “difference” of elements of  $B$ ) is also returned.

**Development(B)**

Let  $B$  be a subset of a magma  $A$  which is a difference set relative to  $A$ , where  $A$  is either the ring  $\mathbb{Z}/m\mathbb{Z}$ , a finite abelian group or an arbitrary finite group (with an iterator). This function constructs the symmetric design having point set  $A$  and whose blocks consist of the sets obtained by translating  $B$  by each element of  $A$  in turn.

**Development(T)**

Let  $T = \{B_1, \dots, B_l\}$  be a difference family consisting of subsets of a magma  $A$  which is either the ring  $\mathbb{Z}/m\mathbb{Z}$ , a finite abelian group or an arbitrary finite group (with an iterator). This function constructs the incidence structure with point set  $A$  and whose  $i$ -th block is the set  $\{B_1 \cup \dots \cup B_l\}$  translated by the  $i$ -th element of  $A$ .

**Example H147E5**

The set  $\{1, 3, 4, 5, 9\}$ , where the elements are residues modulo 11, forms an  $(11, 5, 2)$  difference set. We develop this set and construct a 2- $(11, 5, 2)$  design.

```
> Z11 := IntegerRing(11);
> B := { Z11 | 1, 3, 4, 5, 9};
> IsDifferenceSet(B);
true 2
> D := Development(B);
> D: Maximal;
2-(11, 5, 2) Design with 11 blocks
Points: {@ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 @}
Blocks:
  {1, 3, 4, 5, 9},
  {2, 4, 5, 6, 10},
  {0, 3, 5, 6, 7},
  {1, 4, 6, 7, 8},
  {2, 5, 7, 8, 9},
  {3, 6, 8, 9, 10},
  {0, 4, 7, 9, 10},
  {0, 1, 5, 8, 10},
  {0, 1, 2, 6, 9},
  {1, 2, 3, 7, 10},
```

$\{0, 2, 3, 4, 8\}$

We now construct the twin primes (type “T”) difference set modulo 323 ( $= 17 \times 19$ ), and its development.

```
> B := DifferenceSet(17, "T");
> D := Development(B);
> D;
2-(323, 161, 80) Design with 323 blocks
```

---

## 147.5 Elementary Invariants of an Incidence Structure

All operations defined for incidence structures apply also to near-linear spaces, linear spaces and designs.

**NumberOfPoints(D)**

**#P**

The cardinality  $v$  of the point set  $P$  of the incidence structure  $D$ .

**Points(D)**

An indexed set  $E$  whose elements are the points of the incidence structure  $D$ . Note that this creates a standard set and not the point-set of  $D$ , in contrast to the function **PointSet**.

**Support(D)**

An indexed set  $E$  which is the underlying point set of the incidence structure  $D$  (i.e., the elements of the set have their “real” types; they are no longer from the category **IncPt**).

**PointDegrees(D)**

A sequence whose  $i$ -th term gives the number of blocks containing the  $i$ -th point of the design  $D$ .

**NumberOfBlocks(D)**

**#B**

The number of blocks  $b$  of the incidence structure  $D$  with block-set  $B$ .

**Blocks(D)**

An indexed set containing the blocks of the incidence structure  $D$ . In contrast to the function **BlockSet**, this function returns the collection of blocks of  $D$  in the form of a standard set.

**BlockDegrees(D)**

**BlockSizes(D)**

A sequence whose  $i$ -th term gives the number of points in the  $i$ -th block of the incidence structure  $D$ .

**Covalence(D, S)**

Given a subset  $S$  of the point set of an incidence structure  $D$ , return the number of blocks of  $D$  that contain  $S$ .

**IncidenceMatrix(D)**

The incidence matrix of the incidence structure  $D$ .

**pRank(D, p)**

The  $p$ -rank of the incidence structure  $D$ .

## 147.6 Elementary Invariants of a Design

The following functions can be applied only to designs.

**Parameters(D)**

The parameters  $t$ – $(v, b, r, k, \lambda)$  of the design  $D$  returned as a record.

**ReplicationNumber(D)**

The number of blocks  $r$  containing any point of the  $t$ – $(v, k, \lambda)$  design  $D$ , where  $t > 0$ .

**BlockDegree(D)**

**BlockSize(D)**

The number of points in a block of the design  $D$ .

**Covalence(D, s)**

Given a  $t$ – $(v, k, \lambda)$  design  $D$  and an integer  $s$  such that  $0 \leq s \leq t$ , return the value of  $\lambda_s$ ; i.e., the number of blocks that contain an arbitrary  $s$ -subset of the points of  $D$ .

**Order(D)**

The order of the  $t$ – $(v, k, \lambda)$  design  $D$ . This is defined only for designs with  $t \geq 2$ .

**IntersectionNumber(D, i, j)**

The block intersection number  $\lambda_i^j$ ; i.e., the number of blocks of the design  $D$  containing an  $i$ -set and disjoint from a  $j$ -set. The arguments  $i$  and  $j$  must satisfy  $i + j \leq t$ .

## PascalTriangle(D)

The “Pascal triangle” of the design  $D$ , returned as a sequence; the  $i$ -th element of the sequence is a sequence representing the  $i$ -th row of the triangle. That is, the  $i$ -th element of the sequence is

$$[\lambda_0^{i-1}, \lambda_1^{i-2}, \dots, \lambda_{i-1}^0].$$

If  $D$  is a Steiner  $t$ -design, then the triangle returned has  $k+1$  rows (where  $k$  is the size of a block of  $D$ ); otherwise the triangle has  $t+1$  rows.

**Example H147E6**

---

We illustrate some of the functions of the previous two sections with an example.

```
> F := Design< 2, 7 | {1,2,4}, {1,3,7}, {2,3,5}, {1,5,6}, {3,4,6}, {4,5,7},
>   {2,6,7} >;
> G := IncidenceStructure< 7 | Blocks(F), {1, 3, 7}, {1, 2, 4},
>   {3, 4, 5}, {2, 3, 6}, {2, 5, 7}, {1, 5, 6}, {4, 6, 7} >;
> F;
2-(7, 3, 1) Design with 7 blocks
> G;
Incidence Structure on 7 points with 14 blocks
> Points(G);
{@ 1, 2, 3, 4, 5, 6, 7 @}
> Blocks(F);
{@
  {1, 2, 4},
  {1, 3, 7},
  {2, 3, 5},
  {1, 5, 6},
  {3, 4, 6},
  {4, 5, 7},
  {2, 6, 7}
@}
> IncidenceMatrix(F);
[1 1 0 1 0 0 0]
[1 0 1 0 0 0 1]
[0 1 1 0 1 0 0]
[1 0 0 0 1 1 0]
[0 0 1 1 0 1 0]
[0 0 0 1 1 0 1]
[0 1 0 0 0 1 1]
> P := Points(F);
> P, Universe(P);
{@ 1, 2, 3, 4, 5, 6, 7 @}
Point-set of 2-(7, 3, 1) Design with 7 blocks
> S := Support(F);
> S, Universe(S);
```

```

{@ 1, 2, 3, 4, 5, 6, 7 @}
Integer Ring
> Covalence(G, {1, 2});
2
> Order(F);
2
> PascalTriangle(F);
      7
     4 3
    2 2 1
   0 2 0 1

```

---

### 147.7 Operations on Points and Blocks

In incidence structures, particularly simple ones, blocks are basically sets. For this reason, the elementary set operations such as `join`, `meet` and `subset` have been made to work on blocks. However, blocks are not true MAGMA enumerated sets, and so the functions `Set` and `Support` below have been provided to convert a block to an enumerated set of points for other uses.

`p in B`

Returns **true** if point  $p$  lies in block  $B$ , otherwise **false**.

`p notin B`

Returns **true** if point  $p$  does not lie in block  $B$ , otherwise **false**.

`S subset B`

Given a subset  $S$  of the point set of the incidence structure  $D$  and a block  $B$  of  $D$ , return **true** if the subset  $S$  of points lies in  $B$ , otherwise **false**.

`S notsubset B`

Given a subset  $S$  of the point set of the incidence structure  $D$  and a block  $B$  of  $D$ , return **true** if the subset  $S$  of points does not lie in  $B$ , otherwise **false**.

`PointDegree(D, p)`

The number of blocks of the incidence structure  $D$  that contain the point  $p$ .

`BlockDegree(D, B)`

`BlockSize(D, B)`

`#B`

The number of points contained in the block  $B$  of the incidence structure  $D$ .

Set(*B*)

The set of points contained in the block *B*.

Support(*B*)

The set of underlying points contained in the block *B* (i.e., the elements of the set have their “real” types; they are no longer from the category IncPt).

IsBlock(*D*, *S*)

Returns **true** iff the set (or block) *S* represents a block of the incidence structure *D*. If **true**, also returns one such block.

Line(*D*, *p*, *q*)

Block(*D*, *p*, *q*)

A block of the incidence structure *D* containing the points *p* and *q* (if one exists). In linear spaces, such a block exists and is unique (assuming *p* and *q* are different).

ConnectionNumber(*D*, *p*, *B*)

The connection number  $c(p, B)$ ; i.e., the number of blocks joining *p* to *B* in the incidence structure *D*.

### Example H147E7

---

The following examples uses some of the functions of the previous section.

```
> D, P, B := Design< 2, 7 | {3, 5, 6, 7}, {2, 4, 5, 6}, {1, 4, 6, 7},
> {2, 3, 4, 7}, {1, 2, 5, 7}, {1, 2, 3, 6}, {1, 3, 4, 5} >;
> D: Maximal;
2-(7, 4, 2) Design with 7 blocks
Points: {@ 1, 2, 3, 4, 5, 6, 7 @}
Blocks:
    {3, 5, 6, 7},
    {2, 4, 5, 6},
    {1, 4, 6, 7},
    {2, 3, 4, 7},
    {1, 2, 5, 7},
    {1, 2, 3, 6},
    {1, 3, 4, 5}
> P.1 in B.1;
false
> P.1 in B.3;
true
> {P| 1, 2} subset B.5;
true
> Block(D, P.1, P.2);
{1, 2, 5, 7}
> b := B.4;
```

```

> b;
{2, 3, 4, 7}
> b meet {2, 8};
{ 2 }
> S := Set(b);
> S, Universe(S);
{ 2, 3, 4, 7 }
Point-set of 2-(7, 4, 2) Design with 7 blocks
> Supp := Support(b);
> Supp, Universe(Supp);
{ 2, 3, 4, 7 }
Integer Ring

```

---

## 147.8 Elementary Properties of Incidence Structures and Designs

### Testing the $t$ -balance: A general Note on Parameter A1

A parameter A1 is provided to allow users to specify the algorithm to be used for balance testing. The default value is "NoOrbits", which applies a "brute force" test. One alternative is "Orbits", which uses the orbits of  $t$ -sets under the automorphism group of the incidence structure under consideration. This is much faster than "NoOrbits" for some cases, but slower for others.

The other alternative for checking if an object is  $t$ -balanced is "FastBalanceTest": It is also a "brute force" test but whose implementation allows for a dramatic improvement in efficiency, especially for larger  $t$  ( $t \geq 4$ ). This implementation has a drawback however as it may necessitate more memory space than is available, thus resulting in an out-of-memory error. It is for this reason that "FastBalanceTest" is not the default setting for A1, but as a general rule we strongly recommend its usage as we surmise that execution should successfully complete in most cases.

IsSimple(D)

Returns **true** if and only if the incidence structure  $D$  has no repeated blocks.

IsTrivial(D)

Returns **true** if and only if the incidence structure  $D$  is a trivial incidence structure.

IsSelfDual(D)

Returns **true** if and only if the incidence structure  $D$  is self-dual, that is, if  $D$  is isomorphic to its dual.

IsUniform(D)

Returns **true** if and only if the incidence structure  $D$  is uniform; that is, each block contains the same number of points. If **true**, also returns the blocksize.

**IsNearLinearSpace(D)**

Returns **true** if and only if the incidence structure  $D$  is a near-linear space.

**IsLinearSpace(D)**

Returns **true** if and only if the incidence structure  $D$  is a linear space.

**IsDesign(D, t: parameters)**

**A1**

**MONSTGELT**

*Default : “NoOrbits”*

Returns **true** if and only if the incidence structure  $D$  is a  $t$ -design. If **true**, then the number of blocks of  $D$  containing a general  $t$ -set is also returned. The optional parameter **A1** can be used to specify the algorithm used for balance testing, see the introduction to Section 147.8 for a full description of its usage.

**IsBalanced(D, t: parameters)**

**A1**

**MONSTGELT**

*Default : “NoOrbits”*

Returns **true** if and only if the incidence structure  $D$  is balanced (with respect to  $t$ ). If **true**, then the number of blocks of  $D$  containing a general  $t$ -set is also returned. The optional parameter **A1** can be used to specify the algorithm used for balance testing, see the introduction to Section 147.8 for a full description of its usage.

**IsComplete(D)**

Return **true** if and only if  $D$  is the complete design.

**IsSymmetric(D)**

Return **true** if and only if the design  $D$  is symmetric.

**IsSteiner(D, t)**

**A1**

**MONSTG**

*Default : “NoOrbits”*

Return **true** if and only if the design  $D$  is a Steiner  $t$ -design. The optional parameter **A1** can be used to specify the algorithm used for balance testing, see the introduction to Section 147.8 for a full description of its usage.

**IsPointRegular(D)**

Return **true** if and only if the (near-)linear space  $D$  is point regular. If **true**, the point regularity is also returned.

**IsLineRegular(D)**

Return **true** if and only if the (near-)linear space  $D$  is line regular. If **true**, the line regularity is also returned.



## 147.9 Resolutions, Parallelisms and Parallel Classes

Let  $D$  be an incidence structure with  $v$  points. A *resolution* of  $D$  is a partition of the blocks of  $D$  into classes  $C_i$  such that each class is a 1-design with  $v$  points and index  $\lambda$ . The positive integer  $\lambda$  is called the *index* of the resolution. A resolution with  $\lambda = 1$  is called a *parallelism*. In this case the classes  $C_i$  are called *parallel classes*.

All the functions which deal with resolutions apply to general incidence structures. The functions `HasParallelism`, `AllParallelisms`, `HasParallelClass`, `IsParallelClass` and `AllParallelClasses` require that the incidence structure be uniform. The function `IsParallelism` however applies to a general incidence structure.

**HasResolution(D)**

Return **true** if and only if the incidence structure  $D$  has a resolution. If **true**, one resolution is returned as the second value of the function and the index of the resolution is returned as the third value.

**HasResolution(D,  $\lambda$ )**

Return **true** if and only if the incidence structure  $D$  has a resolution with index  $\lambda$ . If **true**, one such resolution is returned as the second value of the function.

**AllResolutions(D)**

Returns all resolutions of the incidence structure  $D$ .

**AllResolutions(D,  $\lambda$ )**

Returns all resolutions of the incidence structure  $D$  having index  $\lambda$ . When the problem is to find all parallelisms ( $\lambda = 1$ ) in a uniform design  $D$ , it is best to use the function `AllParallelisms` described below.

**IsResolution(D, P)**

Returns **true** if and only if the set  $P$  of blocks (or sets) is a resolution of the incidence structure  $D$ . If **true**, also returns the index of the resolution as the second value of the function.

**HasParallelism(D: *parameters*)**

**A1**

**MONSTG**

*Default* : “Backtrack”

Returns **true** if and only if the uniform incidence structure  $D$  has a parallelism. If **true**, a parallelism is returned as the second value of the function. The parameter **A1** allows the user to specify the algorithm used.

**A1 := "Backtrack"**: A backtrack search is employed. This is in general very efficient when the design  $D$  is parallelizable, irrespective of the number of blocks. In particular, it compares very favourably with the algorithm **Clique**, described below. The later algorithm is recommended when the design appears to have no parallelism. Backtrack is the default.

**A1 := "Clique"**: This algorithm proceeds in two stages. Assume that  $D$  has  $v$  points and  $b$  blocks of size  $k$ . Define the graph  $G_1$  of  $D$  to be the graph whose vertices are the blocks of  $D$  and where two blocks are adjacent if and only if they

are disjoint. The first stage of the Clique algorithm involves constructing the graph  $G_1$  and finding all cliques of size  $v/k$  in  $G_1$ . Any such clique is a parallel class of  $D$ . The second stage involved building a graph  $G_2$  of  $D$  where the vertices are the parallel classes of  $D$  and two parallel classes are adjacent if and only if they are disjoint. The second stage of the Clique algorithm involves searching for cliques of size  $b/(v/k)$  in  $G_2$ . If such a clique exists then it yields a parallelism of  $D$ . (This algorithm was communicated by Vladimir Tonchev).

The clique algorithm is recommended when the design is suspected of having no parallelism. As an example, in the case of a non-parallelizable design with 272 blocks, Clique took around 0.1 second to complete as compared to 2 seconds for the backtrack algorithm. For a non-parallelizable design having 6642 blocks, Clique took around four hours to complete as compared to Backtrack which did not complete in 15 hours. Apart from performing very poorly when  $D$  has a parallelism, Clique may require a considerable amount of memory. It is recommended that Backtrack be tried first, and if it does not complete in a reasonable amount of time, then Clique should be used.

#### AllParallelisms(D)

Returns all parallelisms of the uniform incidence structure  $D$ . This function is to be preferred to the function `AllResolutions(D, 1)` when  $D$  is uniform. This is the case since the implementation of `AllParallelClasses` implies finding cliques in graphs (see the algorithm Clique described in the function `HasParallelism`), while `AllResolutions` performs a full backtrack search.

#### IsParallelism(D, P)

Returns `true` if and only if the set  $P$  of blocks (or sets) is a parallelism of the incidence structure  $D$ .

#### HasParallelClass(D)

Returns `true` if and only if the uniform incidence structure  $D$  has a parallel class.

#### IsParallelClass(D, B, C)

Returns `true` if and only if the uniform incidence structure  $D$  has a parallel class containing the blocks  $B$  and  $C$ . If such a parallel class does exist, one is returned as the second value of the function.

#### AllParallelClasses(D)

Returns all parallel classes of the uniform incidence structure  $D$ .

**Example H147E8**

---

Some of the above functions are illustrated here.

```
> D := IncidenceStructure< 6 | {1,2,3}, {4,5,6}, {1,3,4}, {2,5,6}>;
> bool, R, lambda := HasResolution(D);
> bool;
true
> R;
{
  {
    {1, 2, 3},
    {4, 5, 6},
    {1, 3, 4},
    {2, 5, 6}
  }
}
> lambda;
2
> HasResolution(D,2);
true {
  {
    {1, 2, 3},
    {4, 5, 6},
    {1, 3, 4},
    {2, 5, 6}
  }
}
> AllResolutions(D);
[
  {
    {
      {1, 2, 3},
      {4, 5, 6},
      {1, 3, 4},
      {2, 5, 6}
    }
  },
  {
    {
      {1, 2, 3},
      {4, 5, 6}
    },
    {
      {1, 3, 4},
      {2, 5, 6}
    }
  }
]
```

```

> HasParallelism(D);
true {
  {
    {1, 2, 3},
    {4, 5, 6}
  },
  {
    {1, 3, 4},
    {2, 5, 6}
  }
}
> V := PointSet(D);
> S := { PowerSet(PowerSet(V)) |
>       { {1, 2, 3}, {4, 5, 6} }, { {1, 3, 4}, {2, 5, 6} } };
> IsParallelism(D, S);
true
> B := BlockSet(D);
> S := { { B.1, B.2 }, {B.3, B.4 } };
> IsParallelism(D, S);
true
> AllParallelClasses(D);
{
  {
    {1, 2, 3},
    {4, 5, 6}
  },
  {
    {1, 3, 4},
    {2, 5, 6}
  }
}

```

---

## 147.10 Conversion Functions

### IncidenceStructure(I)

Given an incidence structure  $I$  (of any type), return the same structure as a “true” incidence structure (i.e., belonging to the category `Inc`).

### NearLinearSpace(I)

Given an incidence structure  $I$  (of any type), return the same structure as a near-linear space (i.e., belonging to the category `IncNsp`), if possible.

**LinearSpace(I)**

Given an incidence structure  $I$  (of any type), return the same structure as a linear space (i.e., belonging to the category **IncLsp**), if possible.

**Design(I, t)**

Given an incidence structure  $I$  (of any type), return the same structure as a  $t$ -design (i.e., belonging to the category **Dsgn**), if possible.

**Example H147E9**

---

The following example illustrates some of the functions of the previous two sections.

```
> I := IncidenceStructure< 8 | {1, 2, 3, 4}, {1, 3, 5, 6}, {1, 4, 7, 8},
> {2, 5, 6, 7}, {2, 5, 7, 8}, {3, 4, 6, 8} >;
> IsDesign(I, 1);
true 3
> IsDesign(I, 2);
false
> D := Design(I, 1);
> D;
1-(8, 4, 3) Design with 6 blocks
> IsSteiner(D, 1);
false
> IsNearLinearSpace(I);
false
```

---

**147.11 Identity and Isomorphism**

Incidence structures  $D_1 = (P_1, B_1, I_1)$  and  $D_2 = (P_2, B_2, I_2)$  are *identical* if  $P_1 = P_2$ ,  $B_1 = B_2$  and  $I_1 = I_2$ .

**D eq E**

Return **true** if the incidence structures  $D$  and  $E$  are identical, otherwise **false**.

**D ne E**

Return **true** if the incidence structures  $D$  and  $E$  are not identical, otherwise **false**.

IsIsomorphic(D, E: <i>parameters</i> )
--

AutomorphismGroups

MONSTGELT

Default : "None"

Return **true** if the incidence structures  $D$  and  $E$  are isomorphic, otherwise **false**. If they are isomorphic, then an isomorphism is returned as the second value. The function first computes none, one, or both of the automorphism groups of the left and right incidence structures. In difficult examples, this may significantly speed up testing for isomorphism. The parameter **AutomorphismGroups**, with valid string values "Both", "Left", "Right", "None", may be used to specify which of the automorphism groups should be constructed if not already known. The default is "None".

## 147.12 The Automorphism Group of an Incidence Structure

The automorphism group  $A$  of an incidence structure  $D$  is always presented as a permutation group  $G$  acting on the standard support. The reasons for this include the fact that the support chosen when computing the automorphism group is often quite complicated. Further, if the group is represented as acting on a set of objects relating to the incidence structure, printed permutations are often unreadable. The support used when constructing the automorphism group of an incidence structure  $D$  is either the point set of  $D$  (when  $D$  is simple), or the disjoint union of the point set with the block set (when  $D$  has repeated blocks).

The automorphism group  $G$  of  $D$  does not act directly on  $D$ . Instead,  $G$ -sets are used to transfer the action of  $G$  to various sets associated with  $D$ . The two most important  $G$ -sets, corresponding to action of  $G$  on the point set and on the block set, are returned by each of the functions provided for constructing automorphism group or some specified subgroup of it.

In some circumstances, rather than viewing automorphisms as group elements, it is desirable to view them as mappings of  $D$  into itself. Associated with each incidence structure is a mapping structure,  $\text{Aut}(D)$ , which denotes the set of automorphisms of  $D$ . Note that  $\text{Aut}(D)$  is the *parent* of the automorphisms of  $G$  so that the function  $\text{Aut}(D)$  simply creates a shell structure rather than the actual automorphism group of  $D$ . A transfer map is provided to convert a permutation of the automorphism group  $G$  into a mapping belonging to  $\text{Aut}(D)$ .

### 147.12.1 Construction of Automorphism Groups

AutomorphismGroup(D)
----------------------

Construct the automorphism group  $G$  of the incidence structure  $D$ . The set on which  $G$  acts depends upon whether or not the design is simple. Suppose  $D$  has  $v$  points and  $b$  blocks. If the incidence structure  $D$  is simple, the automorphism group is constructed in its action on the points of  $D$ . It is returned acting on the standard support  $\{1, \dots, v\}$ , where  $i$  corresponds to the  $i$ -th point of  $D$ . If  $D$  is not simple, it is constructed in its action of the disjoint union of the point set and block set of  $D$ .

Again, it is returned acting on the standard support  $\{1, \dots, v+b\}$ , where  $1 \leq i \leq v$  corresponds to the  $i$ -th point of  $D$  and  $v+1 \leq i \leq v+b$  corresponds to the  $(i-v)$ -th point of  $D$ . The function returns:

- (i) The automorphism group  $G$  of  $D$  as described above;
- (ii) A  $G$ -set  $Y$  corresponding to the action of  $G$  on the point set of  $D$ .
- (iii) A  $G$ -set  $W$  corresponding to the action of  $G$  on the block set of  $D$ .
- (iv) The *Aut* structure  $S$  for  $D$ ; and
- (v) A transfer map  $t : G \rightarrow S$ .

Given a permutation  $g$  of  $G$ ,  $t(g)$  is the *mapping* of  $D$  into itself that corresponds to the automorphism group element  $g$ . The  $G$ -sets  $Y$  and  $W$  should be used whenever it is necessary to have  $G$  act on the points or lines of  $D$ .

#### AutomorphismSubgroup(D)

A cyclic subgroup  $H$  of the automorphism group  $G$  of the incidence structure  $D$ . The purpose of this function is to terminate the search for automorphisms of  $D$  as soon as a non-trivial automorphism is found. The function returns:

- (i) The automorphism group  $H$  of  $D$  as described above;
- (ii) The *Aut* structure  $S$  for  $D$ ; and
- (iii) A transfer map  $t : G \rightarrow S$ .

Given a permutation  $g$  of  $H$ ,  $t(g)$  is the *mapping* of  $D$  into itself that corresponds to the automorphism group element  $g$ .

#### AutomorphismGroupStabilizer(D, k)

The subgroup  $H$  of the automorphism group  $G$  of the incidence structure  $D$ , which stabilizes the first  $k$  base points of  $G$ . This function is provided so as to return a subgroup of  $G$  that is sometimes easier to compute than all of  $G$ . The function returns:

- (i) The automorphism group  $H$  of  $D$  as described above;
- (ii) The *Aut* structure  $S$  for  $D$ ; and
- (iii) A transfer map  $t : G \rightarrow S$ .

Given a permutation  $g$  of  $H$ ,  $t(g)$  is the *mapping* of  $D$  into itself that corresponds to the automorphism group element  $g$ .

#### PointGroup(D)

The automorphism group of the incidence structure  $D$  given in its action on the point set of  $D$ , together with the points  $G$ -set.

#### BlockGroup(D)

The automorphism group of the incidence structure  $D$  given in its action on the block set of  $D$ .

Aut( $D$ )

The power structure  $A$  of all automorphisms of the incidence structure  $D$ , together with the transfer map  $t : \text{Sym}(n) \rightarrow A$ , where the points of  $\text{Sym}(n)$  are assumed to be in one-to-one correspondence with the natural support for the automorphism group of  $D$ .

**Example H147E10**

We construct a  $3 - (16, 8, 3)$  Hadamard design and investigate its automorphism group. The first step is to construct a Hadamard matrix of order 16.

```
> R := MatrixRing(Integers(), 4);
> H := R ! [1,1,1,-1, 1,1,-1,1, 1,-1,1,1, -1,1,1,1];
> L := TensorProduct(H, -H);
> D, P, B := HadamardRowDesign(L, 1);
> D;
3-(16, 8, 3) Design with 30 blocks
> G, pg, bg, A, t := AutomorphismGroup(D);
> G;
Permutation group G acting on a set of cardinality 16
Order = 322560 = 2^10 * 3^2 * 5 * 7
(9, 15)(10, 16)(11, 13)(12, 14)
(5, 11, 7, 9)(6, 12, 8, 10)(13, 15)(14, 16)
(5, 9, 14)(6, 10, 13)(7, 11, 16)(8, 12, 15)
(1, 2)(7, 8)(9, 13, 10, 14)(11, 16, 12, 15)
(2, 5)(4, 7)(10, 13)(12, 15)
(3, 9)(4, 10)(7, 13)(8, 14)
(3, 8, 15, 12)(4, 7, 16, 11)(5, 9)(6, 10)
(5, 6)(7, 8)(13, 14)(15, 16)
(3, 12, 4, 11)(7, 16, 8, 15)(9, 10)(13, 14)
(3, 7)(4, 8)(11, 15)(12, 16)
(3, 8)(4, 7)(9, 10)(11, 15)(12, 16)(13, 14)
(9, 10)(11, 12)(13, 14)(15, 16)
(9, 13)(10, 14)(11, 15)(12, 16)
> CompositionFactors(G);
G
| Alternating(8)
*
| Cyclic(2)
*
| Cyclic(2)
*
| Cyclic(2)
*
| Cyclic(2)
1
```

---



### 147.12.2 Action of Automorphisms

As noted at the beginning of the section, the automorphism group  $G$  of an incidence structure  $D$  is given in its action on the standard support and it does not act directly on  $D$ . The action of  $G$  on  $D$  is obtained using the  $G$ -set mechanism. The two basic  $G$ -sets associated with  $D$  correspond to the action of  $G$  on the set of points  $P$  and the set of blocks  $B$  of  $D$ . These two  $G$ -sets are given as return values of the function **AutomorphismGroup** or may be constructed directly. Additional  $G$ -sets associated with  $D$  may be built using the  $G$ -set constructors. Given a  $G$ -set  $Y$  for  $G$ , the action of  $G$  on  $Y$  may be studied using the permutation group functions that allow a  $G$ -set as an argument. In this section, only a few of the available functions are described: see the section on  $G$ -sets for a complete list.

**Image( $g$ ,  $Y$ ,  $y$ )**

Let  $G$  be a subgroup of the automorphism group for the incidence structure  $D$  and let  $Y$  be a  $G$ -set for  $G$ . Given an element  $y$  belonging either to  $Y$  or to a  $G$ -set derived from  $Y$ , find the image of  $y$  under  $G$ .

**Orbit( $G$ ,  $Y$ ,  $y$ )**

Let  $G$  be a subgroup of the automorphism group for the incidence structure  $D$  and let  $Y$  be a  $G$ -set for  $G$ . Given an element  $y$  belonging either to  $Y$  or to a  $G$ -set derived from  $Y$ , construct the orbit of  $y$  under  $G$ .

**Orbits( $G$ ,  $Y$ )**

Let  $G$  be a subgroup of the automorphism group for the incidence structure  $D$  and let  $Y$  be a  $G$ -set for  $G$ . This function constructs the orbits of the action of  $G$  on  $Y$ .

**Stabilizer( $G$ ,  $Y$ ,  $y$ )**

Let  $G$  be a subgroup of the automorphism group for the incidence structure  $D$  and let  $Y$  be a  $G$ -set for  $G$ . Given an element  $y$  belonging either to  $Y$  or to a  $G$ -set derived from  $Y$ , construct the stabilizer of  $y$  in  $G$ .

**Action( $G$ ,  $Y$ )**

Given a subgroup  $G$  of the automorphism group of the incidence structure  $D$ , and a  $G$ -set  $Y$  for  $G$ , construct the homomorphism  $\phi : G \rightarrow L$ , where the permutation group  $L$  gives the action of  $G$  on the set  $Y$ . The function returns:

- (a) The natural homomorphism  $\phi : G \rightarrow L$ ;
- (b) The induced group  $L$ ;
- (c) The kernel of the action (a subgroup of  $G$ ).

**ActionImage( $G$ ,  $Y$ )**

Given a subgroup  $G$  of the automorphism group of the incidence structure  $D$ , and a  $G$ -set  $Y$  for  $G$ , construct the permutation group  $L$  giving the action of  $G$  on the set  $Y$ .

ActionKernel( $G$ ,  $Y$ )

Given a subgroup  $G$  of the automorphism group of the incidence structure  $D$ , and a  $G$ -set  $Y$  for  $G$ , construct the kernel of the action of  $G$  on the set  $Y$ .

IsPointTransitive( $D$ )

Returns **true** iff the automorphism group of the incidence structure  $D$  acts transitively on the point set of  $D$ .

IsBlockTransitive( $D$ )

Returns **true** iff the automorphism group of the incidence structure  $D$  acts transitively on the block set of  $D$ .

---

### Example H147E11

In the following example, the automorphism group of the Witt design on 12 points is constructed.

```
> D, P, B := WittDesign(12);
> A, PY, BY := AutomorphismGroup(D);
> A;
Permutation group Aut acting on a set of cardinality 12
Order = 95040 = 2^6 * 3^3 * 5 * 11
(1, 2)(5, 8)(6, 11)(10, 12)
(2, 3)(5, 12)(6, 8)(10, 11)
(3, 4)(5, 10)(6, 11)(8, 12)
(4, 7)(5, 8)(6, 12)(10, 11)
(5, 11, 12, 6)(7, 8, 9, 10)
(5, 9, 12, 7)(6, 10, 11, 8)
(5, 12)(6, 11)(7, 9)(8, 10)
> a := A!(1, 2, 3, 4)(5, 6, 12, 11);
> 4^a;
1
> {1, 2, 3}^a;
{ 2, 3, 4 }
> Image(a, BY, Block(D, 3));
{1, 2, 3, 8, 11, 12}
> S2 := SylowSubgroup(A, 2);
> S2;
Permutation group S2 acting on a set of cardinality 12
Order = 64 = 2^6
(1, 11, 9, 4)(6, 10, 12, 7)
(1, 9)(2, 12)(3, 7)(4, 11)(5, 10)(6, 8)
(1, 4)(3, 5)(7, 10)(9, 11)
> Stabilizer(S2, BY, Block(D, 4));
Permutation group acting on a set of cardinality 12
Order = 2
(1, 9)(3, 5)(6, 7)(10, 12)
> Stabilizer(S2, 6);
Permutation group acting on a set of cardinality 12
```

```

Order = 8 = 2^3
      (1, 4)(3, 5)(7, 10)(9, 11)
      (1, 4, 9, 11)(2, 5, 8, 3)
> 3^S2;
GSet{ 2, 3, 5, 6, 7, 8, 10, 12 }
> IsPointTransitive(D);
true
> IsBlockTransitive(D);
true

```

We calculate the action of the automorphism group of the design on its blocks.

```

> H := ActionImage(A, BY);
> #H;
95040
> H;
Permutation group H acting on a set of cardinality 132
Order = 95040 = 2^6 * 3^3 * 5 * 11

```

---

### 147.13 Incidence Structures, Graphs and Codes

#### IncidenceStructure(G)

Construct the incidence structure  $D$  corresponding to the graph  $G$ , where the blocks of  $D$  correspond to the edges of  $G$ .

#### PointGraph(D)

The point graph  $G$  of the incidence structure  $D$ . The graph  $G$  has the same point set as  $D$  and two vertices  $u$  and  $v$  of  $G$  are adjacent whenever there is a block of  $D$  containing both  $u$  and  $v$ .

#### BlockGraph(D)

The block graph of the incidence structure  $D$ , i.e., the point graph of the dual of  $D$ .

#### IncidenceGraph(D)

The incidence graph of the incidence structure  $D$ . This bipartite graph has as vertex set the union of the point set  $P$  and block set  $B$  of  $D$ . A vertex  $p \in P$  is adjacent to a vertex  $b \in B$  whenever  $p \in b$ .

#### LinearCode(D, K)

Given an incidence structure  $D$  with  $v$  points and a finite field  $K$ , this function returns the linear code  $C$  of length  $v$  generated by the characteristic functions of the blocks of  $D$  considered as vectors of the  $K$ -space  $K^{(v)}$ .

**Example H147E12**

---

The linear code of the Witt 5-(24, 8, 1) design over GF(2) is the extended Golay code over GF(2).

```
> D := WittDesign(24);
> C := LinearCode(D, GF(2));
> C eq GolayCode(GF(2), true);
true
```

---

**147.14 Automorphisms of Matrices**

Matrices may be regarded as defining a design with the entry of a matrix in a particular row and column defining the incidence of the row and column. The automorphism group of a matrix is the set of permutations of the rows and columns of the matrix that leave the matrix unchanged.

$M \sim x$

The action of a permutation on a matrix by permuting rows and columns. If  $M$  has  $r$  rows and  $c$  columns then  $x$  must have degree  $r + c$  and fix the set  $R = \{1..r\}$ . The action of  $x$  on  $R$  gives the permutation of the rows of the matrix. The remainder of  $x$  gives the action on columns.

AutomorphismGroup(M)

Computes the group of all permutations  $x$  such that  $M^x = M$ . Currently this is done by constructing a graph from  $M$ , and applying the AutomorphismGroup function to the graph.

IsIsomorphic(M, N)

Finds a permutation  $x$  such that  $M^x = N$ , if such exists. If a permutation is found return values are true and the permutation, otherwise returns false. Currently this is done by constructing graphs from  $M$  and  $N$ , and applying the IsIsomorphic function to the graphs.

**Example H147E13**

---

We construct a matrix from a Fano polytope and compute its automorphism group.

```
> M := VertexFacetHeightMatrix(PolytopeSmoothFanoDim3(10)); M;
[1 0 0 0 2 1 2 2]
[0 0 0 1 1 2 2 2]
[0 0 1 2 0 2 2 1]
[2 0 1 0 2 0 2 1]
[0 2 0 1 1 2 0 2]
[1 2 0 0 2 1 0 2]
[0 2 1 2 0 2 0 1]
[2 2 1 0 2 0 0 1]
```

```

[2 2 2 1 1 0 0 0]
[1 2 2 2 0 1 0 0]
[1 0 2 2 0 1 2 0]
[2 0 2 1 1 0 2 0]
> A := AutomorphismGroup(M); A;
Permutation group A acting on a set of cardinality 20
Order = 24 = 2^3 * 3
  (2, 4)(3, 12)(5, 8)(7, 9)(13, 18)(15, 16)(17, 20)
  (1, 5)(2, 6)(3, 8)(4, 7)(9, 11)(10, 12)(13, 16)(14, 19)(17, 18)
  (1, 2)(3, 4)(5, 6)(7, 8)(9, 10)(11, 12)(13, 16)(17, 18)
> Nrows(M), Ncols(M);
12, 8
> Orbits(A);
[
  GSet{@ 14, 19 @},
  GSet{@ 13, 18, 16, 17, 15, 20 @},
  GSet{@ 1, 5, 2, 8, 6, 4, 3, 7, 12, 9, 10, 11 @}
]
```

The automorphism group is transitive on the rows of the matrix. Now dualize the matrix and test for isomorphism.

```

> D := Matrix(12, 8, [2-x:x in Eltseq(M)]);
> f, x := IsIsomorphic(M, D); f, x;
true (2, 4)(3, 12)(5, 8)(7, 9)(14, 19)(15, 17)(16, 20)
> M^x eq D;
true
```

## 147.15 Bibliography

[Hal86] Marshall Hall. *Combinatorial Theory*. New York: Wiley, 2nd edition, 1986.



# 148 HADAMARD MATRICES

<b>148.1 Introduction . . . . .</b>	<b>4909</b>	<b>148.5 Databases . . . . .</b>	<b>4912</b>
<b>148.2 Equivalence Testing . . . .</b>	<b>4909</b>	HadamardDatabase()	4912
IsHadamard(H)	4909	SkewHadamardDatabase()	4912
HadamardNormalize(H)	4909	Matrix(D, n, k)	4912
HadamardCanonicalForm(H)	4909	Matrices(D, n)	4912
HadamardInvariant(H)	4909	DegreeRange(D)	4912
IsHadamardEquivalent(H, J : -)	4909	Degrees(D)	4912
HadamardMatrixToInteger(H)	4910	NumberOfMatrices(D, n)	4912
HadamardMatrixFromInteger(x, n)	4910	<i>148.5.1 Updating the Databases . . . .</i>	<i>4913</i>
<b>148.3 Associated 3-Designs . . .</b>	<b>4911</b>	HadamardDatabaseInformation(D : -)	4914
HadamardRowDesign(H, i)	4911	HadamardDatabaseInformationEmpty(: -)	4914
HadamardColumnDesign(H, i)	4911	UpdateHadamardDatabase(~R, S : -)	4914
<b>148.4 Automorphism Group . . .</b>	<b>4912</b>	WriteHadamardDatabase(S, ~R)	4914
HadamardAutomorphismGroup(H : -)	4912	WriteRawHadamardData(S, R)	4915
		SetVerbose("HadamardDB", v)	4915





# Chapter 148

## HADAMARD MATRICES

### 148.1 Introduction

A *Hadamard matrix* is an  $n \times n$  matrix, all of whose entries are  $\pm 1$ , such that every pair of rows and every pair of columns differ in exactly  $\frac{n}{2}$  places. Two such matrices are considered equivalent if one can be transformed into the other by performing row swaps, column swaps, row negations or column negations. The problem of deciding whether two Hadamard matrices are equivalent is hard.

MAGMA contains several specialised routines for working with Hadamard matrices; of special note is the introduction of a canonical form for such matrices (based on Brendan McKay's *nauty* program). This leads to a much faster equivalence algorithm than previously available.

### 148.2 Equivalence Testing

IsHadamard(H)

Returns `true` if and only if the matrix  $H$  is a Hadamard matrix.

HadamardNormalize(H)

Given a Hadamard matrix  $H$ , returns a normalized matrix equivalent to  $H$ . This matrix is created by negating rows and columns to ensure that the first row and column consist entirely of ones.

HadamardCanonicalForm(H)

Given a Hadamard matrix  $H$ , returns a Hadamard-equivalent matrix  $H'$  and transformation matrices  $X$  and  $Y$  such that  $H' = XHY$ . The matrix  $H'$  is canonical in the sense that all matrices that are Hadamard-equivalent to  $H$  (and no others) will produce the same matrix  $H'$ .

HadamardInvariant(H)

Returns a sequence  $S$  of integers giving the 4-profile of the Hadamard matrix  $H$ . All Hadamard-equivalent matrices have the same 4-profile, but so may some inequivalent ones. Thus this test may determine inequivalence of two matrices more cheaply than performing a full equivalence test, but cannot establish equivalence.

IsHadamardEquivalent(H, J : *parameters*)

**A1**

MONSTGEILT

Default : "*nauty*"

Returns `true` if and only if the Hadamard matrices  $H$  and  $J$  are Hadamard equivalent. The parameter **A1** may be set to either "**Leon**" or "**nauty**" (the default). If the "**nauty**" option is chosen and the matrices are equivalent then transformation matrices  $X$  and  $Y$  are also returned, such that  $J = XHY$ .

<b>HadamardMatrixToInteger(H)</b>
-----------------------------------

Returns an integer that encodes the entries in the given Hadamard matrix in more compact form. The intended use is to save time when testing for equality against the same set of matrices many times.

<b>HadamardMatrixFromInteger(x, n)</b>
--

Returns a Hadamard matrix of degree  $n$  whose encoded form is the integer  $x$ . This function is the inverse of `HadamardMatrixToInteger()`.

**Example H148E1**

We demonstrate the use of some of these functions on certain degree 16 Hadamard matrices. For convenience, we create them from the more compact integer form.

```
> S := [
>   47758915483058652629300889924143904114798786457834842901517979108472281628672,
>   52517743516350345514775635367035460577743730272737765852723792818755052170805,
>   69809270372633075610047556428719374057869882804054059134346034969950931648512,
>   7209801227548712796135507135820555403251560090614832684136782016680445345792
> ];
> T := [ HadamardMatrixFromInteger(x, 16) : x in S ];
> &and [ IsHadamard(m) : m in T ];
true
```

Now we can test them for equivalence; we start by checking the 4-profiles.

```
> [ HadamardInvariant(m) : m in T ];
[
  [ 1680, 0, 140 ],
  [ 1680, 0, 140 ],
  [ 1344, 448, 28 ],
  [ 1344, 448, 28 ]
]
```

We see that the only possible equivalencies are between the first two and the last two, since equivalent matrices must have the same 4-profile.

```
> equiv,X,Y := IsHadamardEquivalent(T[1], T[2]);
> equiv;
true
> T[2] eq X*T[1]*Y;
true
> equiv,X,Y := IsHadamardEquivalent(T[3], T[4]);
> equiv;
false
```

So we have three inequivalent matrices. An alternative way to determine the inequivalent matrices would have been to use the canonical forms.

```
> #{ HadamardCanonicalForm(m) : m in T };
```

### 148.3 Associated 3-Designs

HadamardRowDesign( $H$ ,  $i$ )

Given an  $n \times n$  Hadamard matrix  $H$  (with  $n \geq 4$ ) and an integer  $i$  with  $1 \leq i \leq n$ , returns the Hadamard 3-design corresponding to the  $i$ th row of  $H$ .

HadamardColumnDesign( $H$ ,  $i$ )

Given an  $n \times n$  Hadamard matrix  $H$  (with  $n \geq 4$ ) and an integer  $i$  with  $1 \leq i \leq n$ , returns the Hadamard 3-design corresponding to the  $i$ th column of  $H$ .

#### Example H148E2

There is only one Hadamard equivalence class of  $8 \times 8$  Hadamard matrices. We construct one matrix, and two of its designs.

```
> R := MatrixRing(Integers(), 8);
> H := R![1, 1, 1, 1, 1, 1, 1, 1,
>         1, 1, 1, 1, -1, -1, -1, -1,
>         1, 1, -1, -1, 1, 1, -1, -1,
>         1, 1, -1, -1, -1, -1, 1, 1,
>         1, -1, 1, -1, 1, -1, -1, 1,
>         1, -1, 1, -1, -1, 1, 1, -1,
>         1, -1, -1, 1, -1, 1, -1, 1,
>         1, -1, -1, 1, 1, -1, 1, -1];
> IsHadamard(H);
true
> DR := HadamardRowDesign(H, 3);
> DR: Maximal;
3-(8, 4, 1) Design with 14 blocks
Points: {@ 1, 2, 3, 4, 5, 6, 7, 8 @}
Blocks:
  {1, 2, 5, 6},
  {1, 2, 7, 8},
  {1, 2, 3, 4},
  {1, 4, 5, 7},
  {1, 4, 6, 8},
  {1, 3, 6, 7},
  {1, 3, 5, 8},
  {3, 4, 7, 8},
  {3, 4, 5, 6},
  {5, 6, 7, 8},
  {2, 3, 6, 8},
  {2, 3, 5, 7},
```

```

    {2, 4, 5, 8},
    {2, 4, 6, 7}
> HadamardColumnDesign(H, 8);
3-(8, 4, 1) Design with 14 blocks

```

---

## 148.4 Automorphism Group

**HadamardAutomorphismGroup**( $H$  : *parameters*)

**A1**

MONSTGELT

*Default* : “nauty”

Given a Hadamard matrix  $H$  of degree  $n$ , returns the automorphism group of  $H$  as a permutation group of degree  $4n$ . The parameter **A1** may be set to either “Leon” or “nauty” (the default).

## 148.5 Databases

MAGMA contains two databases of Hadamard matrices; the first database includes all inequivalent matrices of degree at most 28, and examples of matrices of all degrees up to 256. The representatives used are the canonical forms (as output by **HadamardCanonicalForm**), and matrices of a given degree are ordered lexicographically (with 1 considered to be less than  $-1$  for the purposes of this ordering).

A database of skew-symmetric Hadamard matrices also exists; in this case the matrices are not stored in canonical form, since the canonical forms are not skew-symmetric. With the exception of the creation routines, the intrinsics below apply to both databases.

**HadamardDatabase**()

Returns the database of Hadamard matrices.

**SkewHadamardDatabase**()

Returns the database of skew-symmetric Hadamard matrices.

**Matrix**( $D$ ,  $n$ ,  $k$ )

Returns the  $k$ th matrix of degree  $n$  in the database  $D$ .

**Matrices**( $D$ ,  $n$ )

Returns the sequence of all matrices of degree  $n$  that are stored in the database  $D$ .

**DegreeRange**( $D$ )

Returns the smallest and largest degrees of matrices in the database  $D$ .

**Degrees**( $D$ )

Returns the sequence of degrees for which there is at least one matrix of that degree in the database  $D$ .

**NumberOfMatrices**( $D$ ,  $n$ )

Return the number of matrices of degree  $n$  in the database  $D$ .

**Example H148E3**

---

```

> D := HadamardDatabase();
> Matrix(D, 16, 3);
[ 1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1]
[ 1  1  1  1  1  1  1  1 -1 -1 -1 -1 -1 -1 -1 -1]
[ 1  1  1  1 -1 -1 -1 -1  1  1  1  1 -1 -1 -1 -1]
[ 1  1 -1 -1  1  1 -1 -1  1  1 -1 -1  1  1 -1 -1]
[ 1  1 -1 -1 -1  1  1 -1  1 -1 -1  1 -1 -1  1  1]
[ 1  1 -1 -1  1 -1 -1  1 -1  1  1 -1 -1 -1  1  1]
[ 1  1 -1 -1 -1 -1  1  1 -1 -1  1  1  1  1 -1 -1]
[ 1 -1 -1  1 -1 -1  1  1  1  1 -1 -1  1 -1 -1  1]
[ 1 -1 -1  1  1  1 -1 -1 -1 -1  1  1  1 -1 -1  1]
[ 1 -1  1 -1  1 -1  1 -1  1 -1  1 -1  1 -1  1 -1]
[ 1 -1  1 -1  1 -1 -1  1  1 -1 -1  1 -1  1 -1  1]
[ 1 -1  1 -1 -1  1  1 -1 -1  1  1 -1 -1  1 -1  1]
[ 1 -1  1 -1 -1  1 -1  1 -1  1 -1  1  1 -1  1 -1]
[ 1 -1 -1  1  1 -1  1 -1 -1  1 -1  1 -1  1  1 -1]
[ 1 -1 -1  1 -1  1 -1  1  1 -1  1 -1 -1  1  1 -1]
[ 1  1  1  1 -1 -1 -1 -1 -1 -1 -1 -1  1  1  1  1]
> NumberOfMatrices(D, 20);
3
> Degrees(D);
[ 1, 2, 4, 8, 12, 16, 20, 24, 28, 32, 36, 40, 44, 48, 52, 56, 60, 64, 68, 72,
76, 80, 84, 88, 92, 96, 100, 104, 108, 112, 116, 120, 124, 128, 132, 136, 140,
144, 148, 152, 156, 160, 164, 168, 172, 176, 180, 184, 188, 192, 196, 200, 204,
208, 212, 216, 220, 224, 228, 232, 236, 240, 244, 248, 252, 256 ]
> [ NumberOfMatrices(D, n) : n in Degrees(D) ];
[ 1, 1, 1, 1, 1, 5, 3, 60, 487, 23, 218, 20, 500, 55, 562, 2, 3, 1, 2, 1, 2, 1,
3, 2, 1, 1, 1, 1, 2, 2, 1, 3, 1, 1, 2, 2, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2, 1, 1, 1,
1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1 ]

```

---

**148.5.1 Updating the Databases**

The databases are noticeably incomplete (for example, it is known that more than 60 000 Hadamard matrices exist for degree 32 whereas only 23 are in the database; also, only matrices of degrees 36, 44, or 52 are present in the skew database). The MAGMA group welcomes contributions of matrices not equivalent to ones already present and will add them to future versions of the databases.

The functions in this section can be used to create new versions of the databases. This allows users to include matrices not already present without having to wait for new official versions of the databases. These functions use a record whose format is not described, as this record should only be manipulated by these functions.

<b>HadamardDatabaseInformation</b> ( <i>D : parameters</i> )
--

**Canonical**

BOOLELT

*Default : true*

This takes an existing Hadamard database and extracts the information in it to an internal form which will be used by the other intrinsics. This internal form is returned. The parameter **Canonical** indicates whether the entries in the database are known to be the canonical forms or not. It defaults to **true**, which is correct for the standard database; it should be set to **false** if working with the skew database or a user-created database where the entries are not canonical.

Note that the value of the **Canonical** parameter also controls whether the database created from this data should store the canonical forms or the original ones. If you want to extract the matrices from a canonical database but store them in a non-canonical one, you should create the non-canonical database first — using either this intrinsic or **HadamardDatabaseInformationEmpty** — with **Canonical** set to **false** and then add the matrices from the database with **UpdateHadamardDatabase** and **Canonical** set to **true**.

<b>HadamardDatabaseInformationEmpty</b> (: <i>parameters</i> )
--

**Canonical**

BOOLELT

*Default : true*

Returns the internal data that would correspond to an empty database. The parameter **Canonical** is used to indicate whether the entries in the database should be written out in canonical form or not.

This allows the creation of a new database, or a slice of an existing one, without including the entirety of a previous one.

<b>UpdateHadamardDatabase</b> (~ <i>R</i> , <i>S : parameters</i> )
---

**Canonical**

BOOLELT

*Default : false*

This augments the record *R* containing the database information with the matrices in the sequence *S*. The matrices are only added if they are inequivalent with matrices already there. Note that this requires finding the canonical forms, which can be expensive. If the matrices in *S* are known to be in canonical form already then the parameter **Canonical** should be set to **true**.

If the matrices of matching degree already in *R* are not known to be canonical then their canonical forms must also be computed. Again, this can take significant time. This is particularly troublesome for the skew database, where the canonical forms are not stored. For one way to deal with this, see the description of **WriteRawHadamardData** below.

<b>WriteHadamardDatabase</b> ( <i>S</i> , ~ <i>R</i> )
--

This creates the database files *name.dat* and *name.ind* from the database data *R*, where *name* is taken from the string *S*. Since canonical forms may need to be computed during this process, the data is passed with a variable reference so that this computation is not lost. (For instance, if one writes out the database, then adds some more matrices and writes it out again.)

WriteRawHadamardData(*S*, *R*)

This saves the data in *R* to the file whose name is given by the string *S*. When loaded, this file will define a single variable **data** which will behave identically to *R*. This is desirable for the non-canonical databases since the canonical forms will not have to be recomputed.

This routine destroys the original contents, if any, of the file.

SetVerbose("HadamardDB", *v*)

This procedure changes the verbose printing level for the Hadamard database update routines. The verbose value *v* should be an integer in the range 0 to 3. This can provide reassuring indications of progress when a long update process is underway.

### Example H148E4

---

Here are some examples of how one might use these routines. We assume that the file **matrixfile** contains code that will define a sequence *S* of Hadamard matrices when loaded.

```
> D := HadamardDatabase();
> #D;
2004
> data := HadamardDatabaseInformation(D);
> SetVerboseLevel("HadamardDB", 1);
> load "matrixfile";
> UpdateHadamardDatabase(~data, S);
3 new matrices added
> WriteHadamardDatabase("~/data/hadamard", ~data);
Sorting matrices by canonical forms
Sorting by degrees
Computing size information
Writing data file
Data file written (2007 matrices total)

Writing index file
Index file written
```

This has created the files **hadamard.dat** and **hadamard.ind** (in the directory **~/data**) containing the updated data for the Hadamard database. The file stems should be **hadamard** for the standard database, or **hadamard\_skew** for the database of skew-symmetric matrices; in this case we wanted the former. In order to use these files instead of the standard database, the library root must be changed. Database files are looked for in the **data** subdirectory of the library root, so in this example the library root should be changed to **~** in order to get the newly created database.

```
> SetLibraryRoot("~");
> D := HadamardDatabase();
> #D;
```

2007

Of course, the library root is used by other databases, so it would be a good idea to set it back to its original value after creating the modified database. A better approach would be to encapsulate this process in a function (which could be put into the user startup file).

```
> function MyHadamardDatabase()
>   oldlibroot := GetLibraryRoot();
>   SetLibraryRoot("~/");
>   D := HadamardDatabase();
>   SetLibraryRoot(oldlibroot);
>   return D;
> end function;
>
> D := MyHadamardDatabase();
> #D;
2007
```

If working with the database of skew-symmetric Hadamard matrices, it will probably be desirable to use the raw data instead, since the computation of the canonical forms takes a lot of time (about 23 minutes on a 750MHz machine). This does have to be done, but by using the raw forms it will only have to be done once, rather than each time.

```
> D := SkewHadamardDatabase();
> #D;
638
> data := HadamardDatabaseInformationEmpty(: Canonical := false);
> for n in Degrees(D) do
>   UpdateHadamardDatabase(~data, Matrices(D, n));
> end for;
> WriteRawHadamardData("skewraw.m", data);
```

Now that the raw data has been created, it can be used in other sessions to update the database (or the same session, of course).

```
> load "skewraw.m";
> load "matrixfile";
> UpdateHadamardDatabase(~data, S);
> WriteHadamardDatabase("~/data/hadamard_skew", ~data);
> WriteRawHadamardData("skewraw.m", data);
```

Note that the updated raw form is saved as well as the database. Now the new data can be accessed in the same way as the previous database was.

```
> SetLibraryRoot("~/");
> D := SkewHadamardDatabase();
> #D;
641
```

As before, for maximum convenience access to the new database should also be put into a function in the startup file.

---



# 149 GRAPHS

<b>149.1 Introduction . . . . .</b>	<b>4921</b>		
<b>149.2 Construction of Graphs and Digraphs . . . . .</b>	<b>4922</b>		
149.2.1 Bounds on the Graph Order . .	4922		
GraphSizeInBytes(n, m : -)	4922		
149.2.2 Construction of a General Graph	4923		
Graph< >	4923		
Graph< >	4923		
IncidenceGraph(A)	4924		
149.2.3 Construction of a General Digraph	4926		
Digraph< >	4926		
Digraph< >	4926		
IncidenceDigraph(A)	4927		
149.2.4 Operations on the Support . . .	4928		
Support(G)	4928		
Support(V)	4928		
ChangeSupport(G, S)	4928		
ChangeSupport(~G, S)	4928		
StandardGraph(G)	4928		
149.2.5 Construction of a Standard Graph	4929		
BipartiteGraph(m, n)	4929		
CompleteGraph(n)	4930		
KCubeGraph(n : -)	4930		
MultipartiteGraph(Q)	4930		
EmptyGraph(n : -)	4930		
NullGraph( : -)	4930		
PathGraph(n : -)	4930		
PolygonGraph(n : -)	4930		
RandomGraph(n, r : -)	4930		
RandomTree(n : -)	4930		
149.2.6 Construction of a Standard Digraph . . . . .	4931		
CompleteDigraph(n)	4931		
EmptyDigraph(n : -)	4931		
RandomDigraph(n, r : -)	4931		
<b>149.3 Graphs with a Sparse Representation . . . . .</b>	<b>4932</b>		
HasSparseRep(G)	4933		
HasDenseRep(G)	4933		
HasSparseRepOnly(G)	4933		
HasDenseRepOnly(G)	4933		
HasDenseAndSparseRep(G)	4933		
<b>149.4 The Vertex-Set and Edge-Set of a Graph . . . . .</b>	<b>4934</b>		
149.4.1 Introduction . . . . .	4934		
149.4.2 Creating Edges and Vertices . .	4934		
EdgeSet(G)	4934		
Edges(G)	4934		
VertexSet(G)	4934		
Vertices(G)	4934		
!	4934		
.	4934		
Index(v)	4935		
!	4935		
!	4935		
.	4935		
149.4.3 Operations on Vertex-Sets and Edge-Sets . . . . .	4936		
#	4936		
in	4936		
notin	4936		
subset	4936		
notsubset	4936		
eq	4936		
eq	4936		
ne	4936		
ne	4936		
ParentGraph(S)	4936		
ParentGraph(s)	4936		
Random(S)	4937		
Representative(S)	4937		
Rep(S)	4937		
for x in S do ... end for;	4937		
for random x in S do ... end for;	4937		
149.4.4 Operations on Edges and Vertices	4937		
EndVertices(e)	4937		
InitialVertex(e)	4937		
TerminalVertex(e)	4937		
IncidentEdges(u)	4937		
<b>149.5 Labelled, Capacitated and Weighted Graphs . . . . .</b>	<b>4938</b>		
<b>149.6 Standard Constructions for Graphs . . . . .</b>	<b>4938</b>		
149.6.1 Subgraphs and Quotient Graphs	4938		
sub< >	4938		
quo< >	4939		
149.6.2 Incremental Construction of Graphs . . . . .	4940		
+	4940		
+=	4941		
AddVertex(~G)	4941		
AddVertices(~G, n)	4941		
AddVertex(~G, l)	4941		
AddVertices(~G, n, L)	4941		
-	4941		
-	4941		
-=	4941		
-=	4941		
RemoveVertex(~G, v)	4941		
RemoveVertices(~G, U)	4941		

+	4941	149.9.1 Graphs Constructed from Groups	4947
+	4941	CayleyGraph(A)	4947
+	4942	CayleyGraph(A : parameter)	4947
+	4942	UnlabelledCayleyGraph(A)	4947
+=	4942	SchreierGraph(A, B)	4948
+=	4942	UnlabelledSchreierGraph(A, B)	4948
+=	4942	OrbitalGraph(P, u, T)	4948
+=	4942	ClosureGraph(P, G)	4948
AddEdge(G, u, v)	4942	PaleyGraph(q)	4948
AddEdge(G, u, v, l)	4942	PaleyTournament(q)	4948
AddEdge(~G, u, v)	4942	149.9.2 Graphs Constructed from Designs	4948
AddEdge(~G, u, v, l)	4942	IncidenceGraph(D)	4948
AddEdges(G, S)	4942	PointGraph(D)	4949
AddEdges(G, S, L)	4942	BlockGraph(D)	4949
AddEdges(~G, S)	4943	IncidenceGraph(P)	4949
AddEdges(~G, S, L)	4943	PointGraph(P)	4949
-	4943	LineGraph(P)	4949
-	4943	HadamardGraph(H : -)	4949
-	4943	149.9.3 Miscellaneous Graph	
-=	4943	Constructions . . . . .	4949
-=	4943	Converse(G)	4949
-=	4943	OddGraph(n)	4949
-=	4943	TriangularGraph(n)	4950
RemoveEdge(~G, e)	4943	SquareLatticeGraph(n)	4950
RemoveEdges(~G, S)	4943	ClebschGraph()	4950
RemoveEdge(~G, u, v)	4943	ShrikhandeGraph()	4950
149.6.3 Constructing Complements, Line		GewirtzGraph()	4950
Graphs; Contraction, Switching .	4943	ChangGraphs()	4950
Complement(G)	4943	149.10 Elementary Invariants of a	
Contract(e)	4944	Graph . . . . .	4950
Contract(u, v)	4944	Order(G)	4950
Contract(S)	4944	NumberOfVertices(G)	4950
InsertVertex(e)	4944	Size(G)	4950
InsertVertex(T)	4944	NumberOfEdges(G)	4950
LineGraph(G)	4944	CharacteristicPolynomial(G)	4950
Switch(u)	4944	Spectrum(G)	4950
Switch(S)	4944	149.11 Elementary Graph Predicates	4951
149.7 Unions and Products of		adj	4951
Graphs . . . . .	4945	adj	4951
Union(G, H)	4945	notadj	4951
join	4945	notadj	4951
EdgeUnion(G, H)	4946	in	4952
CompleteUnion(G, H)	4946	notin	4952
CartesianProduct(G, H)	4946	eq	4952
LexProduct(G, H)	4946	IsSubgraph(G, H)	4952
TensorProduct(G, H)	4946	IsBipartite(G)	4952
~	4946	IsComplete(G)	4952
149.8 Converting between Graphs		IsEulerian(G)	4952
and Digraphs . . . . .	4947	IsForest(G)	4953
OrientatedGraph(G)	4947	IsEmpty(G)	4953
UnderlyingGraph(D)	4947	IsNull(G)	4953
UnderlyingDigraph(G)	4947	IsPath(G)	4953
149.9 Construction from Groups,		IsPolygon(G)	4953
Codes and Designs . . . . .	4947	IsRegular(G)	4953
		IsTree(G)	4953

**149.12 Adjacency and Degree . . . 4953***149.12.1 Adjacency and Degree Functions  
for a Graph . . . . . 4953*

Degree(u)	4953
Alldeg(G, n)	4953
MaximumDegree(G)	4953
Maxdeg(G)	4953
MinimumDegree(G)	4954
Mindeg(G)	4954
DegreeSequence(G)	4954
Valence(G)	4954
Neighbours(u)	4954
Neighbors(u)	4954
IncidentEdges(u)	4954
Bipartition(G)	4954
MinimumDominatingSet(G)	4954

*149.12.2 Adjacency and Degree Functions  
for a Digraph . . . . . 4954*

InDegree(u)	4954
OutDegree(u)	4954
Degree(u)	4955
Alldeg(G, n)	4955
MaximumInDegree(G)	4955
Maxindeg(G)	4955
MaximumOutDegree(G)	4955
Maxoutdeg(G)	4955
MinimumInDegree(G)	4955
Minindeg(G)	4955
MinimumOutDegree(G)	4955
Minoutdeg(G)	4955
MaximumDegree(G)	4955
Maxdeg(G)	4955
MinimumDegree(G)	4955
Mindeg(G)	4955
DegreeSequence(G)	4956
InNeighbours(u)	4956
InNeighbors(u)	4956
OutNeighbours(u)	4956
OutNeighbors(u)	4956
IncidentEdges(u)	4956

**149.13 Connectedness . . . . . 4956***149.13.1 Connectedness in a Graph . . . . . 4956*

IsConnected(G)	4956
Components(G)	4956
Component(u)	4956
IsSeparable(G)	4956
IsBiconnected(G)	4956
CutVertices(G)	4957
Bicomponents(G)	4957

*149.13.2 Connectedness in a Digraph . . . . . 4957*

IsStronglyConnected(G)	4957
IsWeaklyConnected(G)	4957
StronglyConnectedComponents(G)	4957
Component(u)	4957

*149.13.3 Graph Triconnectivity . . . . . 4957*

IsTriconnected(G)	4958
Splitcomponents(G)	4958
SeparationVertices(G)	4958

*149.13.4 Maximum Matching in Bipartite  
Graphs . . . . . 4959*

MaximumMatching(G)	4959
--------------------	------

*149.13.5 General Vertex and Edge Con-  
nectivity in Graphs and Digraphs . . 4960*

VertexSeparator(G)	4960
VertexConnectivity(G)	4960
IsKVertexConnected(G, k)	4961
EdgeSeparator(G)	4961
EdgeConnectivity(G)	4961
IsKEdgeConnected(G, k)	4961

**149.14 Distances, Paths and Circuits  
in a Graph . . . . . 4963***149.14.1 Distances, Paths and Circuits in a  
Possibly Weighted Graph . . . . . 4963*

Reachable(u, v)	4963
Distance(u, v)	4963
Geodesic(u, v)	4963

*149.14.2 Distances, Paths and Circuits in a  
Non-Weighted Graph . . . . . 4963*

Diameter(G)	4963
DiameterPath(G)	4963
Ball(u, n)	4964
Sphere(u, n)	4964
DistancePartition(u)	4964
IsEquitable(G, P)	4964
EquitablePartition(P, G)	4964
Girth(G)	4964
GirthCycle(G)	4964

**149.15 Maximum Flow, Minimum  
Cut, and Shortest Paths . . 4964****149.16 Matrices and Vector Spaces  
Associated with a Graph or Di-  
graph . . . . . 4965**

AdjacencyMatrix(G)	4965
DistanceMatrix(G)	4965
IncidenceMatrix(G)	4965
IntersectionMatrix(G, P)	4965

**149.17 Spanning Trees of a Graph or  
Digraph . . . . . 4965**

SpanningTree(G)	4965
SpanningForest(G)	4965
BreadthFirstSearchTree(u)	4966
BFSTree(u)	4966
DepthFirstSearchTree(u)	4966
DFSTree(u)	4966

**149.18 Directed Trees . . . . . 4966**

IsRootedTree(G)	4966
Root(G)	4966
IsRoot(v)	4966

RootSide(v)	4966	Image(a, Y, y)	4984
VertexPath(u,v)	4967	Orbit(A, Y, y)	4984
BranchVertexPath(u,v)	4967	Orbits(A, Y)	4984
<b>149.19 Colourings . . . . . 4967</b>		Stabilizer(A, Y, y)	4984
ChromaticNumber(G)	4967	Action(A, Y)	4984
OptimalVertexColouring(G)	4967	ActionImage(A, Y)	4985
ChromaticIndex(G)	4967	ActionKernel(A, Y)	4985
OptimalEdgeColouring(G)	4967	<b>149.23 Symmetry and Regularity</b>	
ChromaticPolynomial(G)	4967	<b>Properties of Graphs . . . 4987</b>	
<b>149.20 Cliques, Independent Sets . 4968</b>		IsTransitive(G)	4987
HasClique(G, k)	4969	IsVertexTransitive(G)	4987
HasClique(G, k, m : -)	4969	IsEdgeTransitive(G)	4987
HasClique(G, k, m, f : -)	4969	OrbitsPartition(G)	4987
MaximumClique(G : -)	4970	IsPrimitive(G)	4987
CliqueNumber(G : -)	4970	IsSymmetric(G)	4987
AllCliques(G : -)	4970	IsDistanceTransitive(G)	4987
AllCliques(G, k : -)	4970	IsDistanceRegular(G)	4987
AllCliques(G, k, m : -)	4971	IntersectionArray(G)	4988
MaximumIndependentSet(G: -)	4971	<b>149.24 Graph Databases and Graph</b>	
IndependenceNumber(G: -)	4971	<b>Generation . . . . . 4989</b>	
<b>149.21 Planar Graphs . . . . . 4973</b>		<i>149.24.1 Strongly Regular Graphs . . . . 4989</i>	
IsPlanar(G)	4973	StronglyRegularGraphsDatabase()	4989
Obstruction(G)	4974	Classes(D)	4989
IsHomeomorphic(G : -)	4974	NumberOfClasses(D)	4989
Faces(G)	4974	NumberOfGraphs(D)	4989
Face(u, v)	4974	NumberOfGraphs(D, S)	4989
Face(e)	4974	Graphs(D, S)	4989
NFaces(G)	4974	Graph(D, S, i)	4990
NumberOfFaces(G)	4974	RandomGraph(D)	4990
Embedding(G)	4974	RandomGraph(D, S)	4990
Embedding(v)	4974	for G in D do ... end for;	4990
PlanarDual(G)	4974	<i>149.24.2 Small Graphs . . . . . 4991</i>	
<b>149.22 Automorphism Group of a</b>		SmallGraphDatabase(n : -)	4991
<b>Graph or Digraph . . . . . 4976</b>		EulerianGraphDatabase(n : -)	4991
<i>149.22.1 The Automorphism Group Func-</i>		PlanarGraphDatabase(n)	4991
<i>tion . . . . . 4976</i>		SelfComplementaryGraphDatabase(n)	4991
AutomorphismGroup(G : -)	4976	#	4992
<i>149.22.2 nauty Invariants . . . . . 4977</i>		Graph(D, i)	4992
IsPartitionRefined(G: -)	4979	Random(D)	4992
<i>149.22.3 Graph Colouring and</i>		for G in D do ... end for;	4992
<i>Automorphism Group . . . . . 4979</i>		<i>149.24.3 Generating Graphs . . . . . 4992</i>	
<i>149.22.4 Variants of Automorphism Group</i>	4980	GenerateGraphs(n : -)	4992
CanonicalGraph(G)	4980	NextGraph(F: -)	4993
EdgeGroup(G)	4980	<i>149.24.4 A General Facility . . . . . 4995</i>	
IsIsomorphic(G, H : -)	4980	OpenGraphFile(s, f, p)	4995
<i>149.22.5 Action of Automorphisms . . . 4984</i>		<b>149.25 Bibliography . . . . . 4997</b>	

# Chapter 149

## GRAPHS

### 149.1 Introduction

A *simple graph* is a graph in which each edge joins two distinct vertices and two distinct vertices are joined by at most one edge. A *simple digraph*, whose edges are directed, is defined in an analogous manner. Thus, loops and multiple edges are not permitted in simple graphs and simple digraphs. A graph (digraph) with loops and/or multiple edges joining a fixed pair of vertices is called a *multigraph* (resp. *multidigraph*). A multidigraph whose edges are assigned a capacity is more commonly called a *network*.

In this chapter the term “graph” is used when referring to a simple undirected graph, while the term “digraph” is used when referring to a simple directed graph. Sometimes the term “graph” is used as the generic term for the incidence structure on vertices and edges. Such uses should be clear from the context in which they occur.

There are five MAGMA graph objects: the undirected simple graph of type `GrphUnd`, the directed simple graph of type `GrphDir`, the undirected multigraph of type `GrphMultUnd`, the directed multigraph of type `GrphMultDir`, and the network of type `GrphNet`. The simple graphs are all of type `Grph`, while the multigraphs (including the network) are of type `GrphMult`. There is a caveat here with respect to simple digraphs and loops: for historical reasons, MAGMA allows loops in digraphs.

Simple graphs and digraphs are covered in this chapter, while multigraphs, multidigraphs and networks are covered in Chapter 150. All types of graphs may have vertex labellings and/or edge labellings. In addition, assigning weights and/or capacities to graph edges is also possible, thus allowing to run shortest-paths and flow-based algorithms over the graphs.

Importantly, from the present version (V2.11) onwards, all the standard graph construction functions (see Subsections 149.6.1 and 149.6.2) respect the graph’s support set and vertex and edge decorations. That is, the resulting graph will have a support set and vertex and edge decorations compatible with the original graph and the operation performed on that graph.

MAGMA employs two distinct data structures for representing graphs. A graph may be represented in the form of an adjacency matrix (the dense representation) or in the form of an adjacency list (the sparse representation). The latter is better suited for sparse graphs and for algorithms which have been designed with the adjacency list representation in mind. An advantage of the sparse representation is the possibility of creating much larger (sparse) graphs than would be possible using the dense representation, since memory requirements for the adjacency list representation is linear in the number of edges, while memory requirements for the adjacency matrix representation is quadratic in the number of vertices (order) of the graph. This is covered in detail in Section 149.2.1.

Further, multigraphs and multidigraphs (and networks) may only be represented by an adjacency list since they may contain multiple edges. Users have control over the choice of representation when creating simple graphs or digraphs. If no indication is given, simple graphs and digraphs are *always* created with the dense representation. At the time of the present release (V2.11), a significant part of MAGMA functions are able to work directly with either of the representations. However, wherever necessary, a graph will be converted internally to whichever representation is required by a given function. We emphasize that this process is completely transparent to users and *requires no intervention on their part*.

For a concise and comprehensive overview on graph theory and algorithms, the reader is referred to [Eve79]; in [TCR90] they'll find a clear exposure of some graph algorithms, and [RAO93] is the recommended reference for flow problems in graphs.

## 149.2 Construction of Graphs and Digraphs

Any enumerated or indexed set  $S$  may be given as the vertex-set of a graph. The graph constructor will take a copy  $V$  of  $S$ , convert  $V$  into an indexed set if necessary, and flag its type as `GrphVertSet`. A graph may be specifically created as a sparse graph. If no indication is given then the graph is *always* created with the dense representation, that is, as an adjacency matrix.

### 149.2.1 Bounds on the Graph Order

Memory allocation for any elementary object (that is, accessed via a single pointer) cannot exceed  $2^{32}$  bytes. For this reason there is a bound on the graph order. This bound is dependent upon the (internal) graph representation chosen at creation: the dense representation (as a packed adjacency matrix), or the sparse representation (as an adjacency list). The former is quadratic in the number of vertices of the graph, the latter is linear in the number of the edges. Thus a large graph with a low edge density will require less memory space than the same graph with a dense representation.

The bounds on the graph order  $n$  are as follows:

- for the dense representation,  $n \leq 65535$ ,
- for the sparse representation,  $n \leq 134217722$ .

These bounds are maximal, that is, they assume — for the sparse representation — that the number of edges is zero. To help users determine the likely size of the graph they want to construct, we provide the following function.

`GraphSizeInBytes(n, m : parameters)`

<code>IsDigraph</code>	<code>BOOL</code>	<i>Default : false</i>
<code>SparseRep</code>	<code>BOOL</code>	<i>Default : false</i>

Computes the memory requirement in bytes of a graph of order  $n$  and size  $m$ . By default it is assumed that the graph is undirected and has a dense representation.

**Example H149E1**

One may verify that the dense representation cannot be used for a value of  $n$  (graph order) larger than 65535.

```
> n := 65536;
> m := 0;
> assert GraphSizeInBytes(n, m) gt 2^32;
```

It is possible to construct such a graph with a sparse representation:

```
> GraphSizeInBytes(n, m : SparseRep := true);
2097580
```

However, assuming that the graph of order  $n = 65535$  has a size of  $m = 33538101$  we see that such a graph cannot be constructed as its total memory requirement is again larger than  $2^{32}$ :

```
> m := 33538101;
> assert GraphSizeInBytes(n, m : SparseRep := true) gt 2^32;
```

**149.2.2 Construction of a General Graph**

Graph< n   edges : parameters >
Graph< S   edges : parameters >

**SparseRep****BOOL****Default : false**

Construct the graph  $G$  with vertex-set  $V = \{@v_1, v_2, \dots, v_n@\}$  (where  $v_i = i$  for each  $i$  if the first form of the constructor is used, or the  $i$ th element of the enumerated or indexed set  $S$  otherwise), and edge-set  $E = \{e_1, e_2, \dots, e_q\}$ . This function returns three values: The graph  $G$ , the vertex-set  $V$  of  $G$ ; and the edge-set  $E$  of  $G$ . If **SparseRep** is **true** then the resulting graph will have a sparse representation.

The elements of  $E$  are specified by the list **edges**, where the items of **edges** may be objects of the following types:

- (a) A pair  $\{v_i, v_j\}$  of vertices in  $V$ . The edge from  $v_i$  to  $v_j$  will be added to the edge-set for  $G$ .
- (b) A tuple of the form  $\langle v_i, N_i \rangle$  where  $N_i$  will be interpreted as a set of neighbours for the vertex  $v_i$ . The elements of the sets  $N_i$  must be elements of  $V$ . If  $N_i = \{u_1, u_2, \dots, u_r\}$ , the edges  $\{v_i, u_1\}, \dots, \{v_i, u_r\}$  will be added to  $G$ .
- (c) A sequence  $[N_1, N_2, \dots, N_n]$  of  $n$  sets, where  $N_i$  will be interpreted as a set of neighbours for the vertex  $v_i$ . The edges  $\{v_i, u_i\}$ ,  $1 \leq i \leq n$ ,  $u_i \in N_i$ , are added to  $G$ .

In addition to these three basic ways of specifying the *edges* list, the items in *edges* may also be:

- (d) An edge  $e$  of a graph or digraph or multigraph or multidigraph or network of order  $n$ . If  $e$  is an edge from  $u$  to  $v$  in a directed graph  $H$ , then the undirected edge  $\{u, v\}$  will be added to  $G$ .
- (e) An edge-set  $E$  of a graph or digraph or multigraph or multidigraph or network of order  $n$ . Every edge  $e$  in  $E$  will be added to  $G$ .
- (f) A graph or a digraph or a multigraph or a multidigraph or a network  $H$  of order  $n$ . Every edge  $e$  in  $H$ 's edge-set is added to  $G$ .
- (g) A  $n \times n$  symmetric  $(0, 1)$ -matrix  $A$ . The matrix  $A$  will be interpreted as the adjacency matrix for a graph  $H$  on  $n$  vertices and the edges of  $H$  will be added will be added to  $G$ .
- (h) A set of
  - (i) Pairs of the form  $\{v_i, v_j\}$  of vertices in  $V$ .
  - (ii) Tuples of the form  $\langle v_i, N_i \rangle$  where  $N_i$  will be interpreted as a set of neighbours for the vertex  $v_i$ .
  - (iii) Edges of a graph or digraph or multigraph or multidigraph or network of order  $n$ .
  - (iv) Graphs or digraphs or multigraphs or multidigraphs or networks of order  $n$ .
- (j) A sequence of
  - (i) Tuples of the form  $\langle v_i, N_i \rangle$  where  $N_i$  will be interpreted as a set of neighbours for the vertex  $v_i$ .

IncidenceGraph(A)

Let  $A$  be a  $n \times m$   $(0, 1)$  matrix having exactly two 1s in each column. Then a graph  $G$  of order  $n$  and size  $m$  having  $A$  as its incidence matrix will be constructed. The rows of  $A$  will correspond to the vertices of  $G$  while the columns will correspond to the edges.

---

**Example H149E2**

The Petersen graph will be constructed in various different ways to illustrate the use of the constructor. Firstly, it will be constructed by listing the edges:

```
> P := Graph< 10 | { 1, 2 }, { 1, 5 }, { 1, 6 }, { 2, 3 }, { 2, 7 },
>           { 3, 4 }, { 3, 8 }, { 4, 5 }, { 4, 9 }, { 5, 10 },
>           { 6, 8 }, { 6, 9 }, { 7, 9 }, { 7, 10 }, { 8, 10 } >;
```

We next construct the Petersen graph by listing the neighbours of each vertex:

```
> P := Graph< 10 | [ { 2, 5, 6 }, { 1, 3, 7 }, { 2, 4, 8 }, { 3, 5, 9 },
>           { 1, 4, 10 }, { 1, 8, 9 }, { 2, 9, 10 }, { 3, 6, 10 },
>           { 4, 6, 7 }, { 5, 7, 8 } ] >;
```

We repeat the above construction but now the graph is created as a sparse graph:

```
> PS := Graph< 10 | [ { 2, 5, 6 }, { 1, 3, 7 }, { 2, 4, 8 }, { 3, 5, 9 },
```



```

>          { 1, 4, 10 }, { 1, 8, 9 }, { 2, 9, 10 }, { 3, 6, 10 },
>          { 4, 6, 7 }, { 5, 7, 8 } ] : SparseRep := true >;
> assert PS eq P;

```

Here is yet another way of constructing the sparse graph, from the dense graph:

```

> PS := Graph< 10 | P : SparseRep := true >;
> assert PS eq P;

```

We finally construct the graph in terms of an adjacency matrix:

```

> M := MatrixRing( Integers(), 10 );
> P := Graph< 10 | M! [ 0,1,0,0,1,1,0,0,0,0,
>          1,0,1,0,0,0,1,0,0,0,
>          0,1,0,1,0,0,0,1,0,0,
>          0,0,1,0,1,0,0,0,1,0,
>          1,0,0,1,0,0,0,0,0,1,
>          1,0,0,0,0,0,0,1,1,0,
>          0,1,0,0,0,0,0,0,1,1,
>          0,0,1,0,0,1,0,0,0,1,
>          0,0,0,1,0,1,1,0,0,0,
>          0,0,0,0,1,0,1,1,0,0 ] >;
> P;

```

```

      Graph
Vertex Neighbours

```

```

      1      2 5 6 ;
      2      1 3 7 ;
      3      2 4 8 ;
      4      3 5 9 ;
      5      1 4 10 ;
      6      1 8 9 ;
      7      2 9 10 ;
      8      3 6 10 ;
      9      4 6 7 ;
     10      5 7 8 ;

```

### Example H149E3

A more sophisticated example is the construction of Tutte's 8-cage using the technique described in P. Lorimer [Lor89]. The graph is constructed so that it has  $G = \text{P}\Gamma\text{L}(2, 9)$  in its representation of degree 30 as its automorphism group. The vertices of the graph correspond to the points on which  $G$  acts. The neighbours of vertex 1 are the points lying in the unique orbit  $N_1$  of length 3 of the stabilizer of 1. The edges for vertex  $i$  are precisely the points  $N_1^g$  where  $g$  is an element of  $G$  such that  $1^g = i$ .

```

> G := PermutationGroup< 30 |
>      (1, 2)(3, 4)(5, 7)(6, 8)(9, 13)(10, 12)(11, 15)(14, 19) (16, 23)
>      (17, 22)(18, 21)(20, 27)(24, 29)(25, 28)(26, 30),
>      (1, 24, 28, 8)(2, 9, 17, 22)(3, 29, 19, 15)(4, 5, 21, 25)

```

```
> (6, 18, 7, 16)(10, 13, 30, 11)(12, 14)(20, 23)(26, 27) >;
> N1 := rep{ o : o in Orbits(Stabilizer(G, 1)) | #o eq 3 };
> tutte := Graph< 30 | <1, N1>^G >;
```

### 149.2.3 Construction of a General Digraph

Digraph< n   edges : parameters >
Digraph< S   edges : parameters >

SparseRep

BOOL

Default : false

Construct the digraph  $G$  with vertex-set  $V = \{v_1, v_2, \dots, v_n\}$  (where  $v_i = i$  for each  $i$  if the first form of the constructor is used, or the  $i$ th element of the enumerated or indexed set  $S$  otherwise), and edge-set  $E = \{e_1, e_2, \dots, e_q\}$ . This function returns three values: The graph  $G$ , the vertex-set  $V$  of  $G$ ; and the edge-set  $E$  of  $G$ . If SparseRep is **true** then the resulting graph will have a sparse representation.

The elements of  $E$  are specified by the list **edges**, where the items of **edges** may be objects of the following types:

- (a) A pair  $[v_i, v_j]$  of vertices in  $V$ . The directed edge from  $v_i$  to  $v_j$  will be added to the edge-set for  $G$ .
- (b) A tuple of the form  $\langle v_i, N_i \rangle$  where  $N_i$  will be interpreted as a set of out-neighbours for the vertex  $v_i$ . The elements of the sets  $N_i$  must be elements of  $V$ . If  $N_i = \{u_1, u_2, \dots, u_r\}$ , the directed edges  $[v_i, u_1], \dots, [v_i, u_r]$  will be added to  $G$ .
- (c) A sequence  $[N_1, N_2, \dots, N_n]$  of  $n$  sets, where  $N_i$  will be interpreted as a set of neighbours for the vertex  $v_i$ . The directed edges  $[v_i, u_i]$ ,  $1 \leq i \leq n$ ,  $u_i \in N_i$ , are added to  $G$ .

In addition to these three basic ways of specifying the *edges* list, the items in *edges* may also be:

- (d) An edge  $e$  of a graph or digraph or multigraph or multidigraph or network of order  $n$ . If  $e$  is an edge  $\{u, v\}$  in an undirected graph then both directed edges  $[u, v]$  and  $[v, u]$  are added to  $G$ .
- (e) An edge-set  $E$  of a graph or digraph or multigraph or multidigraph or network of order  $n$ . Every edge  $e$  in  $E$  will be added to  $G$  according to the rule set out for a single edge.
- (f) A graph or a digraph or a multigraph or a multidigraph or a network  $H$  of order  $n$ . Every edge  $e$  in  $H$ 's edge-set is added to  $G$  according to the rule set out for a single edge.
- (g) A  $n \times n$  (0, 1)-matrix  $A$ . The matrix  $A$  will be interpreted as the adjacency matrix for a digraph  $H$  on  $n$  vertices and the edges of  $H$  will be added will be added to  $G$ .
- (h) A set of

- (i) Pairs of the form  $[v_i, v_j]$  of vertices in  $V$ .
- (ii) Tuples of the form  $\langle v_i, N_i \rangle$  where  $N_i$  will be interpreted as a set of out-neighbours for the vertex  $v_i$ .
- (iii) Edges of a graph or digraph or multigraph or multidigraph or network of order  $n$ .
- (iv) Graphs or digraphs or multigraphs or multidigraphs or networks of order  $n$ .
- (j) A sequence of
  - (i) Tuples of the form  $\langle v_i, N_i \rangle$  where  $N_i$  will be interpreted as a set of out-neighbours for the vertex  $v_i$ .

**IncidenceDigraph(A)**

Let  $A$  be a  $n \times m$  matrix such that each column contains at most one entry equal to  $+1$  and at most one entry equal to  $-1$  (all others being zero). Then a digraph  $G$  of order  $n$  and size  $m$  having  $A$  as its incidence matrix will be constructed. The rows of  $A$  will correspond to the vertices of  $G$  while the columns will correspond to the edges. If column  $k$  of  $A$  contains  $+1$  in row  $i$  and  $-1$  in row  $j$ , the directed edge  $v_i v_j$  will be included in  $G$ . If only row  $i$  has a non-zero entry (either  $1$  or  $-1$ ), then the loop  $v_i v_i$  will be included in  $G$ .

**Example H149E4**


---

The digraph  $D$  with vertices  $\{v_1, v_2, v_3, v_4, v_5\}$  and edges  $(v_1, v_2), (v_1, v_3), (v_1, v_4), (v_3, v_2)$ , and  $(v_4, v_3)$  will be constructed, firstly by listing its edges and secondly by giving its adjacency matrix.

```
> D1 := Digraph< 5 | [1, 2], [1, 3], [1, 4],
>                [3, 2], [4, 3] >;
> D1;
Digraph
Vertex Neighbours
1      2 3 4 ;
2      ;
3      2 ;
4      3 ;
5      ;
```

The same digraph with a sparse representation:

```
> D1 := Digraph< 5 | [1, 2], [1, 3], [1, 4],
>                [3, 2], [4, 3] : SparseRep := true>;
```

We next construct the digraph by giving its adjacency matrix:

```
> M := MatrixRing(Integers(), 5);
> D2 := Digraph< 5 |
>      M! [ 0,1,1,1,0,
>           0,0,0,0,0,
>           0,1,0,0,0,
```

```

>          0,0,1,0,0,
>          0,0,0,0,0 ] >;
> IsIsomorphic(D1, D2);
true

```

## 149.2.4 Operations on the Support

Let  $S$  be the set over which the graph  $G$  is defined. The set  $S$  is referred to as the *support* of  $G$ .

Support( $G$ )
----------------

Support( $V$ )
----------------

The indexed set used in the construction of  $G$  (or the graph for which  $V$  is the vertex-set), or the standard set  $\{@1, \dots, n@\}$  if it was not given.

ChangeSupport( $G, S$ )
-------------------------

If  $G$  is a graph having  $n$  vertices and  $S$  is an indexed set of cardinality  $n$ , return a new graph  $H$  equal to  $G$  but whose support is  $S$ . That is,  $H$  is structurally equal to  $G$  and its vertex and edge decorations are the *same* as those for  $G$ .

ChangeSupport( $\sim G, S$ )
------------------------------

The procedural version of the above function.

StandardGraph( $G$ )
----------------------

The graph  $H$  equal to  $G$  but defined on the standard support. That is,  $H$  is structurally equal to  $G$  and its vertex and edge decorations are the same as those for  $G$ .

### Example H149E5

The Odd Graph,  $O_n$ , has as vertices all  $n$ -subsets of a  $2n - 1$  set with two vertices adjacent if and only if they are disjoint. We construct  $O_3$  and then form its standard graph.

```

> V := Subsets({1..5}, 2);
> O3 := Graph< V | { {u,v} : u,v in V | IsDisjoint(u, v) } >;
> O3;
Graph
Vertex  Neighbours

```

```

{ 1, 5 }      { 2, 4 } { 2, 3 } { 3, 4 } ;
{ 2, 5 }      { 1, 3 } { 1, 4 } { 3, 4 } ;
{ 1, 3 }      { 2, 5 } { 2, 4 } { 4, 5 } ;
{ 1, 4 }      { 2, 5 } { 3, 5 } { 2, 3 } ;
{ 2, 4 }      { 1, 5 } { 1, 3 } { 3, 5 } ;
{ 3, 5 }      { 1, 4 } { 2, 4 } { 1, 2 } ;

```

```

{ 2, 3 }      { 1, 5 } { 1, 4 } { 4, 5 } ;
{ 1, 2 }      { 3, 5 } { 3, 4 } { 4, 5 } ;
{ 3, 4 }      { 1, 5 } { 2, 5 } { 1, 2 } ;
{ 4, 5 }      { 1, 3 } { 2, 3 } { 1, 2 } ;

```

```
> Support(O3);
```

```

{@
  { 1, 5 },
  { 2, 5 },
  { 1, 3 },
  { 1, 4 },
  { 2, 4 },
  { 3, 5 },
  { 2, 3 },
  { 1, 2 },
  { 3, 4 },
  { 4, 5 }

```

```
@}
```

```
> S03 := StandardGraph(O3);
```

```
> S03;
```

```
Graph
```

```
Vertex  Neighbours
```

```

1      5 7 9 ;
2      3 4 9 ;
3      2 5 10 ;
4      2 6 7 ;
5      1 3 6 ;
6      4 5 8 ;
7      1 4 10 ;
8      6 9 10 ;
9      1 2 8 ;
10     3 7 8 ;

```

```
> Support(S03);
```

```
{@ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 @}
```

### 149.2.5 Construction of a Standard Graph

Some of the construction functions listed below can also be used to create a graph with a sparse representation. Are concerned only those functions creating relatively sparse graphs.

**BipartiteGraph(m, n)**

The complete bipartite graph,  $K_{m,n}$ , with partite sets of cardinality  $m$  and  $n$ .

**CompleteGraph(*n*)**

The complete graph  $K_n$  on  $n$  vertices.

**KCubeGraph(*n* : *parameters*)**

**SparseRep**

BOOL

*Default* : false

The graph of the  $n$ -dimensional cube,  $Q_n$ . The graph can be created as a sparse graph if so requested.

**MultipartiteGraph(*Q*)**

Given a sequence  $Q$  of positive integers  $m_1, \dots, m_r$ , construct the complete multipartite graph  $K_{m_1, \dots, m_r}$ .

**EmptyGraph(*n* : *parameters*)**

**SparseRep**

BOOL

*Default* : false

The graph on  $n$  vertices with no edges. It can be created as a sparse graph if so requested.

**NullGraph( : *parameters*)**

**SparseRep**

BOOL

*Default* : false

The graph with no vertices and no edges. It can be created as a sparse graph if so requested.

**PathGraph(*n* : *parameters*)**

**SparseRep**

BOOL

*Default* : false

The path graph on  $n$  vertices, i.e. the graph on vertices  $v_1, \dots, v_n$ , with  $v_i$  and  $v_j$  ( $1 \leq i < j \leq n$ ) adjacent if and only if  $j = i + 1$ . The graph can be created as a sparse graph if so requested.

**PolygonGraph(*n* : *parameters*)**

**SparseRep**

BOOL

*Default* : false

Given an integer  $n \geq 3$ , define the polygon graph on  $n$  vertices, i.e. the graph on vertices  $v_1, \dots, v_n$ , with  $v_i$  and  $v_j$  ( $1 \leq i < j \leq n$ ) adjacent if and only if  $j = i + 1$ , or  $i = 1$  and  $j = n$ . The graph can be created as a sparse graph if so requested.

**RandomGraph(*n*, *r* : *parameters*)**

**SparseRep**

BOOL

*Default* : false

A random graph on  $n$  vertices such that the probability that an arbitrary pair of vertices is adjacent is given by the real number  $r$ , where  $r$  lies in the interval  $[0, 1]$ . The graph can be created as a sparse graph if so requested.

**RandomTree(*n* : *parameters*)**

**SparseRep**

BOOL

*Default* : false

A random tree on  $n$  vertices. It can be created as a sparse graph if so requested.

### 149.2.6 Construction of a Standard Digraph

As for standard graphs (Subsection 149.2.5) some of the construction functions listed below can be used to create digraphs with a sparse representation.

CompleteDigraph( $n$ )

The complete symmetric digraph on  $n$  vertices.

EmptyDigraph( $n$  : *parameters*)

SparseRep

BOOL

*Default* : false

The null digraph on  $n$  vertices. It can be created as sparse if so requested.

RandomDigraph( $n$ ,  $r$  : *parameters*)

SparseRep

BOOL

*Default* : false

A random digraph on  $n$  vertices such that the probability that there exists an edge directed from vertex  $u$  to vertex  $v$ , where  $u$  and  $v$  arbitrary vertices, is given by the real number  $r$ , where  $r$  lies in the interval  $[0, 1]$ . The digraph can be created as sparse if so requested.

---

#### Example H149E6

The complete symmetric digraph on 5 vertices is created as follows:

```
> kd5 := CompleteDigraph(5);
```

```
> kd5;
```

```
Digraph
```

```
Vertex Neighbours
```

```
1      2 3 4 5 ;
```

```
2      1 3 4 5 ;
```

```
3      1 2 4 5 ;
```

```
4      1 2 3 5 ;
```

```
5      1 2 3 4 ;
```

We next construct a random digraph on 5 vertices such that the probability that there is an edge directed from an arbitrary vertex to any other arbitrary vertex is 0.75.

```
> rd5 := RandomDigraph(5, 0.75);
```

```
> rd5;
```

```
Digraph
```

```
Vertex Neighbours
```

```
1      2 5 ;
```

```
2      2 3 4 5 ;
```

```
3      2 3 4 5 ;
```

```
4      1 2 3 5 ;
```

```
5      2 3 4 ;
```

---

### 149.3 Graphs with a Sparse Representation

As mentioned in the Introduction 149.1 of this chapter it is possible to construct graphs having a sparse representation. That is, graphs which are represented by means of an adjacency list. This has an obvious advantage for graphs with a low edge density, in that it allows to construct much larger graphs than if they were represented by means of an adjacency matrix (the dense representation). See Section 149.2.1 for more details on this issue.

Another advantage of the sparse representation is for graph algorithms which are linear in the number of edges (the planarity tester and the triconnectivity tester), and more generally, for those algorithms based on the adjacency list representation (the flow-based algorithms and the shortest-paths algorithms). Further, the sparse representation is required when creating multigraphs (see Chapter 150) since they may have multiple edges.

A significant number, but not all, of the existing MAGMA functions have been written for both representations. Should a function designed for a matrix representation of the graph be applied to a graph with a sparse representation, then the graph is automatically converted *without any user intervention*. This is also true when the reverse conversion is required.

Without being exhaustive, we will list here the functions which can deal with both graph representations without having to perform an (internal) conversion.

- Nearly all construction functions (149.2),
- All functions in 149.4, 149.6.1, 149.6.2,
- Some functions in 149.6.3 and 149.7,
- All functions in 149.8,
- The basic invariants and predicates in 149.10 and 149.11,
- Almost all functions in 149.12,
- All functions in 149.13.1 and all but the last in 149.13.2,
- The functions in 149.14.1,
- All functions in 149.17.

The functions in 149.5, 149.13.3, 149.13.4, 149.13.5, and 149.21, and all the functions listed in Chapter 150, deal with a graph as an adjacency list.

The functions in 149.9, 149.14.2, 149.16, 149.18, 149.19, 149.20, 149.22, and 149.23 deal with a graph as an adjacency matrix.

We emphasize again that should a conversion of the graph's representation take place due to internal specifications, the conversion takes place *automatically*, that is, *without user intervention*.

The only problem that may occur is when such a conversion concerns a (very) large graph and thus may possibly result in a lack of memory to create the required representation. This is where users might require control over which internal representation is used so as to minimise the likelihood of both representations coexisting. This is achieved by tuning the creation functions (as described in Section 149.2), and the following functions which determine the nature of a graph's representation.



When conversion of the graph's representation into the alternative one occurs the original representation is not deleted.

HasSparseRep(G)
HasDenseRep(G)
HasSparseRepOnly(G)
HasDenseRepOnly(G)
HasDenseAndSparseRep(G)

This set of functions determine the nature of a graph's representation.

#### Example H149E7

---

We give four examples, each illustrating one of the four possible cases that may arise. First, when the graph's dense representation remains unchanged:

```
> G := Graph<5 | { {1,2}, {2,3}, {3,4}, {4,5}, {5,1} }>;
> HasDenseRepOnly(G);
true
> A := AutomorphismGroup(G);
> HasDenseRepOnly(G);
true
```

The same example using a sparse graph representation:

```
> G := Graph<5 | { {1,2}, {2,3}, {3,4}, {4,5}, {5,1} } : SparseRep := true>;
> HasSparseRepOnly(G);
true
> A := AutomorphismGroup(G);
> HasDenseAndSparseRep(G);
true
```

Next the case when the graph's sparse representation remains unchanged:

```
> G := Graph<5 | { {1,2}, {2,3}, {3,4}, {4,5}, {5,1} } : SparseRep := true>;
> HasSparseRepOnly(G);
true
> IsPlanar(G);
true
> HasSparseRepOnly(G);
true
```

Finally the same example using a dense graph representation:

```
> G := Graph<5 | { {1,2}, {2,3}, {3,4}, {4,5}, {5,1} }>;
> HasDenseRepOnly(G);
true
> IsPlanar(G);
true
> HasDenseAndSparseRep(G);
```

true

---

## 149.4 The Vertex–Set and Edge–Set of a Graph

### 149.4.1 Introduction

Let  $G$  be a graph on  $n$  vertices and  $m$  edges whose vertex-set is  $V = \{v_1, \dots, v_m\}$  and edge-set is  $E = \{e_1, \dots, e_m\}$ . A graph created by MAGMA consists of three objects: the *vertex-set*  $V$ , the *edge-set*  $E$  and the graph  $G$  itself. The vertex-set and edge-set of a graph are *enriched* sets and consequently constitute types. The vertex-set and edge-set are returned as the second and third arguments, respectively, by all functions which create graphs. Alternatively, a pair of functions are provided to extract the vertex-set and edge-set of a graph  $G$ . The main purpose of having vertex-sets and edge-sets as types is to provide a convenient mechanism for referring to vertices and edges of a graph. Here, the functions applicable to vertex-sets and edge-sets are described.

### 149.4.2 Creating Edges and Vertices

**EdgeSet( $G$ )**

Given a graph  $G$ , return the edge-set of  $G$ .

**Edges( $G$ )**

A set  $E$  whose elements are the edges of the graph  $G$ . Note that this creates an indexed set and not the edge-set of  $G$ , in contrast to the function **EdgeSet**.

**VertexSet( $G$ )**

Given a graph  $G$ , return the vertex-set of  $G$ .

**Vertices( $G$ )**

A set  $V$  whose elements are the vertices of the graph  $G$ . In contrast to the function **VertexSet**, this function returns the collection of vertices of  $G$  in the form of an indexed set.

**$V ! v$**

Given the vertex-set  $V$  of the graph  $G$  and an element  $v$  of the support of  $V$ , create the corresponding vertex of  $G$ .

**$V . i$**

Given a vertex-set  $V$  and an integer  $i$  such that  $1 \leq i \leq \#V$ , create the vertex  $v_i$  of  $V$ .

Index( $v$ )

Given a vertex  $v$  of some graph  $G$ , return the index of  $v$  in the (indexed) vertex-set of  $G$ .

$E \text{ ! } \{u, v\}$

Given the edge-set  $E$  of the graph  $G$  and objects  $u, v$  belonging to the support of  $G$ , which correspond to adjacent vertices, create the edge  $uv$  of  $G$ .

$E \text{ ! } [u, v]$

Given the edge-set  $E$  of the digraph  $G$  and objects  $u, v$  belonging to the support of  $G$ , which correspond to adjacent vertices, create the edge  $uv$  of  $G$ .

$E . i$

Given an edge-set  $E$  and an integer  $i$  such that  $1 \leq i \leq \#E$ , create the  $i$ -th edge of  $E$ .

---

#### Example H149E8

The construction of vertices and edges is illustrated using the Odd Graph,  $O_3$ . and then form its standard graph.

```
> S := Subsets({1..5}, 2);
> O3, V, E := Graph< S | { {u,v} : u,v in S | IsDisjoint(u, v) } >;
> VertexSet(O3);
Vertex-set of O3
> Vertices(O3);
{@ { 1, 5 }, { 2, 5 }, { 1, 3 }, { 1, 4 }, { 2, 4 }, { 3, 5 },
{ 2, 3 }, { 1, 2 }, { 3, 4 }, { 4, 5 } @}
> EdgeSet(O3);
Edge-set of O3
> Edges(O3);
{@ {{ 1, 5 }, { 2, 4 }}, {{ 1, 5 }, { 2, 3 }}, {{ 1, 5 }, { 3, 4 }},
{{ 2, 5 }, { 1, 3 }}, {{ 2, 5 }, { 1, 4 }}, {{ 2, 5 }, { 3, 4 }},
{{ 1, 3 }, { 2, 4 }}, {{ 1, 3 }, { 4, 5 }}, {{ 1, 4 }, { 3, 5 }},
{{ 1, 4 }, { 2, 3 }}, {{ 2, 4 }, { 3, 5 }}, {{ 3, 5 }, { 1, 2 }},
{{ 2, 3 }, { 4, 5 }}, {{ 1, 2 }, { 3, 4 }}, {{ 1, 2 }, { 4, 5 }} @}
> u := V!{1, 2};
> u, Type(u);
{1, 2} GrphVert
> Index(u);
8
> x := E!{ {1,2}, {3,4}};
> x, Type(x);
{{ 1, 2 }, { 3, 4 }} GrphEdge
```

---

### 149.4.3 Operations on Vertex-Sets and Edge-Sets

For each of the following operations,  $S$  and  $T$  may be interpreted as either the vertex-set or the edge-set of the graph  $G$ . The variable  $s$  may be interpreted as either a vertex or an edge. The edge-set and vertex-set support all the standard set operations.

**#S**

The cardinality of the set  $S$ .

**s in S**

Return **true** if the vertex (edge)  $s$  lies in the vertex-set (edge-set)  $S$ , otherwise **false**.

**s notin S**

Return **true** if the vertex (edge)  $s$  does not lie in the vertex-set (edge-set)  $S$ , otherwise **false**.

**S subset T**

Return **true** if the vertex-set (edge-set)  $S$  is contained in the vertex-set (edge-set)  $T$ , otherwise **false**.

**S notsubset T**

Return **true** if the vertex-set (edge-set)  $S$  is not contained in the vertex-set (edge-set)  $T$ , otherwise **false**.

**S eq T**

Return **true** if the vertex-set (edge-set)  $S$  is equal to the vertex-set (edge-set)  $T$ .

**s eq t**

Returns **true** if the vertex (edge)  $s$  is equal to the vertex (edge)  $t$ .

**S ne T**

Returns **true** if the vertex-set (edge-set)  $S$  is not equal to the vertex-set (edge-set)  $T$ .

**s ne t**

Returns **true** if the vertex (edge)  $s$  is not equal to the vertex (edge)  $t$ .

**ParentGraph(S)**

Return the graph  $G$  for which  $S$  is the vertex-set (edge-set).

**ParentGraph(s)**

Return the graph  $G$  for which  $s$  is a vertex (edge).

Random( $S$ )

Choose a random element from the vertex-set (edge-set)  $S$ .

Representative( $S$ )

Rep( $S$ )

Choose some element from the vertex-set (edge-set)  $S$ .

for  $x$  in  $S$  do ... end for;

The vertex-set (edge-set)  $S$  may appear as the range in the **for**-statement.

for random  $x$  in  $S$  do ... end for;

The vertex-set (edge-set)  $S$  may appear as the range in the **for random** - statement.

#### 149.4.4 Operations on Edges and Vertices

EndVertices( $e$ )

Given an edge  $e$  belonging to the graph  $G$ , return a set containing the two end-vertices of  $e$ . If  $G$  is a digraph return the two end-vertices in a sequence.

InitialVertex( $e$ )

Given an undirected or directed edge  $e$  from vertex  $u$  to vertex  $v$ , return vertex  $u$ . This is useful in the undirected case since it indicates, where relevant, the direction in which the edge has been traversed.

TerminalVertex( $e$ )

Given an undirected or directed edge  $e$  from vertex  $u$  to vertex  $v$ , return vertex  $v$ . This is useful in the undirected case since it indicates, where relevant, the direction in which the edge has been traversed.

IncidentEdges( $u$ )

Given a vertex  $u$  of a graph  $G$ , return the set of all edges incident with the vertex  $u$ . If  $G$  is directed, then the set consists of all the edges incident into  $u$  and from  $v$ .

## 149.5 Labelled, Capacitated and Weighted Graphs

A vertex labelling of a graph  $G$  is a partial map  $f$  from the vertex-set  $V$  of  $G$  into a set  $L$ . An edge labelling of a graph  $G$  is a partial map  $f$  from the edge-set  $E$  of  $G$  into a set  $L$ . A labelled graph is built by assigning labels successively to vertices or edges after the (unlabelled) graph has been constructed.

Similarly, a capacitated graph  $G$  is a partial map from its edge-set into  $Z^+$ , and a weighted graph  $G$  is a partial map from its edge-set into  $R$ ,  $R$  any ring with a total order. Those two last features are particularly convenient when running shortest-paths and flow algorithms. Edge capacities and edge weights are assigned to edges once the graph has been constructed. Any graph edge may carry a label, together with a capacity and/or a weight.

All the functions for decorating graph vertices and edges are fully documented in Section 150.4 in Chapter 150. A few examples are also given there.

## 149.6 Standard Constructions for Graphs

The two main ways in which to construct a new graph from an old one are either by taking a subgraph of or by modifying the original graph. Another method is to take the quotient graph. When the first two methods are employed, the support set and vertex and edge decorations are retained in the resulting graph.

### 149.6.1 Subgraphs and Quotient Graphs

sub< G   list >
-----------------

Construct the graph  $H$  as a subgraph of  $G$ . The function returns three values: The graph  $H$ , the vertex-set  $V$  of  $H$ ; and the edge-set  $E$  of  $H$ . If  $G$  has a support set and/or if  $G$  has vertex/edge decorations, then these attributes are transferred to the subgraph  $H$ .

The elements of  $V$  and of  $E$  are specified by the list *list* whose items can be objects of the following types:

- (a) A vertex of  $G$ . The resulting subgraph will be the subgraph induced on the subset of  $\text{VertexSet}(G)$  defined by the vertices in *list*.
- (b) An edge of  $G$ . The resulting subgraph will be the subgraph with vertex-set  $\text{VertexSet}(G)$  whose edge-set consists of the edges in *list*.
- (c) A set of
  - (i) Vertices of  $G$ .
  - (ii) Edges of  $G$ .

It is easy to recover the map that maps the vertices of the subgraph to the vertices of the supergraph and vice-versa. We give an example below.

quo< G   P >
--------------

Given a graph  $G$  and a partition  $P$  of the vertex-set  $V(G)$  of  $G$ , construct the quotient graph  $Q$  of  $G$  defined by  $P$ . The partition is given as a set of subsets of  $V(G)$ . Suppose the cells of the partition  $P$  are  $P_1, \dots, P_r$ . The vertices of  $Q$  correspond to the cells  $P_i$ . Vertices  $v_i$  and  $v_j$  of  $Q$  are adjacent if and only if there is an edge in  $G$  joining a vertex in  $P_i$  with a vertex in  $P_j$ .

---

**Example H149E9**

We start with a simple example which demonstrates how to map the vertices of a subgraph onto the vertices of the supergraph and vice-versa: This is achieved by simple coercion into the appropriate vertex-set.

```
> K5, V := CompleteGraph(5);
> K3, V1 := sub< K5 | { V | 3, 4, 5 } >;
> IsSubgraph(K5, K3);
true
>
> V!V1!1;
3
>
> V1!V!4;
2
>
> V1!V!1;
>> V1!V!1;
^
```

Runtime error in '!': Illegal coercion

LHS: GrphVertSet

RHS: GrphVert

A 1-factor of a graph  $G$  is a set of disjoint edges which form a spanning forest for  $G$ . In the following example, we construct the graph that corresponds to  $K_6$  with a 1-factor removed.

```
> K6, V, E := CompleteGraph(6);
> K6;
Graph
Vertex  Neighbours
1      2 3 4 5 6 ;
2      1 3 4 5 6 ;
3      1 2 4 5 6 ;
4      1 2 3 5 6 ;
5      1 2 3 4 6 ;
6      1 2 3 4 5 ;
> F1 := { E | {1,2}, {3, 4}, {5, 6} };
> G1, V1, E1 := K6 - F1;
> G1;
Graph
```

Vertex	Neighbours
--------	------------

1	3 4 5 6 ;
2	3 4 5 6 ;
3	1 2 5 6 ;
4	1 2 5 6 ;
5	1 2 3 4 ;
6	1 2 3 4 ;

**Example H149E10**

Taking the complete graph  $K_9$ , we form its quotient with respect to the partition of the vertex-set into three 3-sets.

```
> K9, V, E := CompleteGraph(9);
> P := { { V | 1, 2, 3 }, { V | 4, 5, 6 }, { V | 7, 8, 9 } };
> Q := quo< K9 | P >;
> Q;
```

Graph

Vertex	Neighbours
--------	------------

1	2 3 ;
2	1 3 ;
3	1 2 ;

**149.6.2 Incremental Construction of Graphs**

Unless otherwise specified, each of the functions described in this section returns three values:

- (i) The graph  $G$ ;
- (ii) The vertex-set  $V$  of  $G$ ;
- (iii) The edge-set  $E$  of  $G$ .

**149.6.2.1 Adding Vertices**

$G + n$
---------

Given a graph  $G$  and a non-negative integer  $n$ , adds  $n$  new vertices to  $G$ . The existing vertex and edge decorations are retained, but the support will become the standard support.



$G \mathrel{+}:= n$
---------------------

<code>AddVertex(<math>\sim G</math>)</code>
---

<code>AddVertices(<math>\sim G</math>, <math>n</math>)</code>
---

The procedural version of the previous function. `AddVertex` adds one vertex only to  $G$ .

<code>AddVertex(<math>\sim G</math>, <math>l</math>)</code>
---

Given a graph  $G$  and a label  $l$ , adds a new vertex with label  $l$  to  $G$ . The existing vertex and edge decorations are retained, but the support will become the standard support.

<code>AddVertices(<math>\sim G</math>, <math>n</math>, <math>L</math>)</code>
---

Given a graph  $G$  and a non-negative integer  $n$ , and a sequence of  $n$  labels, adds  $n$  new vertices to  $G$  with labels from  $L$ . The existing vertex and edge decorations are retained, but the support will become the standard support.

### 149.6.2.2 Removing Vertices

$G - v$
---------

$G - U$
---------

Given a graph  $G$  and a vertex  $v$  or a set of vertices  $U$  of  $G$ , removes  $v$  or the vertices in  $U$  from  $G$ . The support and vertex and edge decorations are retained.

$G \mathrel{-}:= v$
---------------------

$G \mathrel{-}:= U$
---------------------

<code>RemoveVertex(<math>\sim G</math>, <math>v</math>)</code>
--

<code>RemoveVertices(<math>\sim G</math>, <math>U</math>)</code>
--

The procedural versions of the previous functions.

### 149.6.2.3 Adding Edges

$G + \{ u, v \}$
------------------

$G + [ u, v ]$
----------------

Given a graph  $G$  and a pair of vertices of  $G$ , add the edge to  $G$  described by this pair. If  $G$  is undirected then the edge must be given as a set of (two) vertices, if  $G$  is directed the edge is given as a sequence of (two) vertices. The support and existing vertex and edge decorations are retained. This set of functions has two return values: The first is the modified graph and the second is the newly created edge.

$G + \{ \{ u, v \} \}$
$G + \{ [ u, v ] \}$

Given a graph  $G$  and a set of pairs of vertices of  $G$ , add the edges to  $G$  described by these pairs. If  $G$  is undirected then edges must be given as a set of (two) vertices, if  $G$  is directed edges are given as a sequence of (two) vertices. The support and existing vertex and edge decorations are retained.

$G += \{ u, v \}$
$G += [ u, v ]$
$G += \{ \{ u, v \} \}$
$G += \{ [ u, v ] \}$

The procedural versions of the previous four functions.

$\text{AddEdge}(G, u, v)$
---------------------------

Given a graph  $G$ , two vertices of  $G$   $u$  and  $v$ , returns a new edge between  $u$  and  $v$ . The support and existing vertex and edge decorations are retained. This function has two return values: The first is the modified graph and the second is the newly created edge.

$\text{AddEdge}(G, u, v, l)$
------------------------------

Given a graph  $G$ , two vertices of  $G$   $u$  and  $v$ , and a label  $l$ , adds a new edge with label  $l$  between  $u$  and  $v$ . The support and existing vertex and edge decorations are retained. This function has two return values: The first is the modified graph and the second is the newly created edge.

$\text{AddEdge}(\sim G, u, v)$
$\text{AddEdge}(\sim G, u, v, l)$

Procedural versions of previous functions adding edges to a graph.

$\text{AddEdges}(G, S)$
-------------------------

Given a graph  $G$ , a set  $S$  of pairs of vertices of  $G$ , adds the edges specified in  $S$ . The elements of  $S$  must be sets or sequences of two vertices of  $G$ , depending on whether  $G$  is undirected or directed respectively. The support and existing vertex and edge decorations are retained.

$\text{AddEdges}(G, S, L)$
----------------------------

Given a graph  $G$ , a sequence  $S$  of pairs of vertices of  $G$ , and a sequence  $L$  of labels of same length, adds the edges specified in  $S$  with its corresponding label in  $L$ . The elements of  $S$  must be sets or sequences of two vertices of  $G$ , depending on whether  $G$  is undirected or directed respectively. The support and existing vertex and edge decorations are retained.

<code>AddEdges(<math>\sim G</math>, <math>S</math>)</code>
--

<code>AddEdges(<math>\sim G</math>, <math>S</math>, <math>L</math>)</code>
--

Procedural versions of previous functions adding edges to a graph.

#### 149.6.2.4 Removing Edges

<code><math>G - e</math></code>
---------------------------------

<code><math>G - \{ e \}</math></code>
---------------------------------------

Given a graph  $G$  and an edge  $e$  or a set  $S$  of edges of  $G$ , removes  $e$  or the edges in  $S$  from  $G$ . The resulting graph will have vertex-set  $V(G)$  and edge-set  $E(G) \setminus \{e\}$  or  $E(G) \setminus S$ . The support and existing vertex and edge decorations are retained.

<code><math>G - \{ \{ u, v \} \}</math></code>
--

<code><math>G - \{ [u, v] \}</math></code>
--

Given a graph  $G$  and a set  $S$  of pairs  $\{u, v\}$  or  $[u, v]$ ,  $u, v$  vertices of  $G$ , form the graph having vertex-set  $V(G)$  and edge-set  $E(G)$  minus the edges from  $u$  to  $v$  for all pairs in  $S$ . An edge is represented as a set if  $G$  is undirected, as a set otherwise.

The support and existing vertex and edge decorations are retained.

<code><math>G -:= e</math></code>
-----------------------------------

<code><math>G -:= \{ e \}</math></code>
---

<code><math>G -:= \{ \{ u, v \} \}</math></code>
--

<code><math>G -:= \{ [u, v] \}</math></code>
--

<code>RemoveEdge(<math>\sim G</math>, <math>e</math>)</code>
--

<code>RemoveEdges(<math>\sim G</math>, <math>S</math>)</code>
---

<code>RemoveEdge(<math>\sim G</math>, <math>u</math>, <math>v</math>)</code>
--

The procedural versions of the previous functions.

#### 149.6.3 Constructing Complements, Line Graphs; Contraction, Switching

Unless otherwise stated, each of the functions described here apply to both graphs and digraphs. Further, each of the functions returns three values:

- (i) The graph  $G$ ;
- (ii) The vertex-set  $V$  of  $G$ ;
- (iii) The edge-set  $E$  of  $G$ .

<code>Complement(<math>G</math>)</code>
---

The complement of the graph  $G$ .

**Contract( $e$ )**

Given an edge  $e = \{u, v\}$  of the graph  $G$ , form the graph obtained by identifying the vertices  $u$  and  $v$ , and removing any multiple edges (and loops if the graph is undirected) from the resulting graph. The graph's support and vertex/edges decorations are retained.

**Contract( $u, v$ )**

Given vertices  $u$  and  $v$  belonging to the graph  $G$ , return the graph obtained by identifying the vertices  $u$  and  $v$ , and removing any multiple edges (and loops if the graph is undirected) from the resulting graph. The graph's support and vertex/edges decorations are retained.

**Contract( $S$ )**

Given a set  $S$  of vertices belonging to the graph  $G$ , return the graph obtained by identifying all of the vertices in  $S$ , and removing any multiple edges (and loops if the graph is undirected) from the resulting graph. The graph's support and vertex/edges decorations are retained.

**InsertVertex( $e$ )**

Given an edge  $e$  in the graph  $G$ , insert a new vertex of degree 2 in  $e$ . If applicable, the two new edges thus created that replace  $e$  will have the same capacity and weight as  $e$ . They will be unlabelled. The vertex labels and the edge decorations of  $G$  are retained, but the resulting graph will have standard support.

**InsertVertex( $T$ )**

Given an a set  $T$  of edges belonging to the graph  $G$ , insert a vertex of degree 2 in each edge belonging to the set  $T$ . Vertex and edge decorations are handled as in the single edge case.

**LineGraph( $G$ )**

The line graph of the non-empty graph  $G$ .

**Switch( $u$ )**

Given a vertex  $u$  in an undirected graph  $G$ , construct an undirected graph  $H$  from  $G$ , where  $H$  has the same vertex-set as  $G$  and the same edge-set, except that the vertices that were the neighbours of  $u$  in  $G$  become the non-neighbours of  $u$  in  $H$ , and the vertices that were non-neighbours of  $u$  in  $G$  become the neighbours of  $u$  in  $H$ . The support and vertex labels are *not* retained in the resulting graph.

**Switch( $S$ )**

Given a set  $S$  of vertices belonging to the undirected graph  $G$ , apply the function **Switch( $u$ )**, to each vertex of  $S$  in turn. The support and vertex labels are *not* retained in the resulting graph.

**Example H149E11**

---

We illustrate the use of some of these operations by using them to construct the Grötzsch graph. This graph may be built by taking the complete graph  $K_5$ , choosing a cycle of length 5 (say, 1-3-5-2-4), inserting a vertex of degree two on each edge of this cycle and finally connecting each of these vertices to a new vertex.

```
> G := CompleteGraph(5);
> E := EdgeSet(G);
> H := InsertVertex({ E | { 1, 3 }, { 1, 4 }, { 2, 4 }, { 2, 5 }, { 3, 5 } } );
> L := Union(H, CompleteGraph(1));
> V := VertexSet(L);
> L := L + { { V.11, V.6 }, { V.11, V.7 }, { V.11, V.8 }, { V.11, V.9 },
>           { V.11, V.10 } };
> L;
```

Graph

Vertex    Neighbours

```
1         2 5 6 7 ;
2         1 3 8 10 ;
3         2 4 6 9 ;
4         3 5 7 8 ;
5         1 4 9 10 ;
6         1 3 11 ;
7         1 4 11 ;
8         2 4 11 ;
9         3 5 11 ;
10        2 5 11 ;
11        6 7 8 9 10 ;
```

---

## 149.7    Unions and Products of Graphs

The support and vertex/edge decorations of the original graphs are *not* retained in the graph resulting from applying any of the union functions below.

Union(G, H)
-------------

G join H
----------

Given graphs  $G$  and  $H$  with disjoint vertex sets  $V(G)$  and  $V(H)$ , respectively, construct their union, i.e. the graph with vertex-set  $V(G) \cup V(H)$ , and edge-set  $E(G) \cup E(H)$ .

EdgeUnion( $G$ ,  $H$ )

Given graphs  $G$  and  $H$  having the same number of vertices, construct their edge union  $K$ . This construction identifies the  $i$ -th vertex of  $G$  with the  $i$ -th vertex of  $H$  for all  $i$ . The edge union has the same vertex-set as  $G$  (and hence as  $H$ ) and vertices  $u$  and  $v$  of  $K$  are adjacent if and only if either  $u$  and  $v$  are adjacent in  $G$  or  $u$  and  $v$  are adjacent in  $H$ .

CompleteUnion( $G$ ,  $H$ )

Given graphs  $G$  and  $H$  with disjoint vertex-sets  $V(G)$  and  $V(H)$ , respectively, construct the complete union of  $G$  and  $H$ . This graph consists of the union of  $G$  and  $H$  (Union( $G$ ,  $H$ )), together with edges  $uv$ , for all  $u$  in  $V(G)$  and all  $v$  in  $V(H)$ .

CartesianProduct( $G$ ,  $H$ )

Given graphs  $G$  and  $H$  with disjoint vertex-sets  $V(G)$  and  $V(H)$ , respectively, form the product  $K = G \times H$  of  $G$  and  $H$ . The product has vertex-set  $V(G) \times V(H)$ . Two vertices  $u = (u_1, u_2)$  and  $v = (v_1, v_2)$  of  $K$  are adjacent when either

- (a)  $u_1 = v_1$  and  $u_2 \text{ adj } v_2$ , or
- (b)  $u_2 = v_2$  and  $u_1 \text{ adj } v_1$ .

LexProduct( $G$ ,  $H$ )

Given graphs  $G$  and  $H$  with disjoint vertex-sets  $V(G)$  and  $V(H)$ , respectively, form the lexicographic product  $K$  of  $G$  and  $H$ . The lexicographic product has vertex-set  $V(G) \times V(H)$ . Two vertices  $u = (u_1, u_2)$  and  $v = (v_1, v_2)$  of  $K$  are adjacent when either

- (a)  $u_1 \text{ adj } v_1$ , or
- (b)  $u_1 = v_1$  and  $u_2 \text{ adj } v_2$ .

TensorProduct( $G$ ,  $H$ )

Given graphs  $G$  and  $H$  with disjoint vertex-sets  $V(G)$  and  $V(H)$ , respectively, form the tensor product  $K$  of  $G$  and  $H$ . This graph has vertex-set  $V(G) \times V(H)$ . Two vertices  $u = (u_1, u_2)$  and  $v = (v_1, v_2)$  of  $K$  are adjacent when  $u_1 \text{ adj } v_1$  and  $u_2 \text{ adj } v_2$ .

$G \sim n$

Given a graph  $G$  and a positive integer  $n$ , construct the  $n$ -th power  $K$  of  $G$ . This graph has the same vertex-set as  $G$ , and vertices  $u$  and  $v$  of  $K$  are adjacent if and only if the distance between  $u$  and  $v$  in  $G$  is less than or equal to  $n$ .

## 149.8 Converting between Graphs and Digraphs

Note that the two functions `UnderlyingGraph` and `UnderlyingDigraph` may also be used when one needs to get a copy of a graph  $G$  without  $G$ 's support and vertex/edge decorations.

`OrientatedGraph(G)`

Given a graph  $G$ , produce a digraph  $D$  whose vertex-set is the same as that of  $G$  and whose edge-set consists of the edges of  $G$ , each given a direction. The edges of  $D$  are always directed from the lower numbered vertex to the higher numbered vertex. Thus, if  $G$  contains the edge  $\{u, v\}$ , then  $D$  will have the edge  $[u, v]$  if  $u < v$ , otherwise the edge  $[v, u]$ . The support and vertex/edge decorations of  $G$  are not retained.

`UnderlyingGraph(D)`

The underlying graph  $G$  of the graph  $D$ ;  $G$  has the same vertex-set as  $D$ . If  $D$  is undirected, then  $G$  is a copy of  $D$  without  $D$ 's support and vertex/edge decorations. If  $D$  is directed, then two vertices  $u$  and  $v$  are adjacent in  $G$  if and only if, in  $D$ , there is either an edge directed from  $u$  to  $v$  or from  $v$  to  $u$ . The support and vertex/edge decorations of  $G$  are not retained.

`UnderlyingDigraph(G)`

The underlying digraph  $D$  of the graph  $G$ ;  $D$  has the same vertex-set as  $G$ . If  $G$  is directed, then  $D$  is a copy of  $G$  without  $D$ 's support and vertex/edge decorations. If  $G$  is undirected, then if vertices  $u$  and  $v$  are adjacent in  $G$  then, in  $D$ , there will be both an edge directed from  $u$  to  $v$  and an edge directed from  $v$  to  $u$ . The support and vertex/edge decorations of  $G$  are not retained.

## 149.9 Construction from Groups, Codes and Designs

### 149.9.1 Graphs Constructed from Groups

`CayleyGraph(A)`

`CayleyGraph(A : parameter)`

`UnlabelledCayleyGraph(A)`

`Labelled`

`BOOLELT`

`Default : true`

Given a finite group  $A$  defined on generating set  $X$ , construct the Cayley graph  $C$  of  $A$  relative to the generating set  $X$ . This graph is defined as follows: The vertices correspond to the elements of  $A$  and two vertices  $u, v$  are adjacent if and only if there exists an element  $x$  in  $X$  such that  $u * x = v$ .

The optional parameter `Labelled` (`Labelled := true` by default) can be set to `false` to prevent the graph being labelled. If this is not done, then the vertices of

$C$  will be labelled with the appropriate elements of  $A$  and the (directed) edge from  $u$  to  $v$  will be labelled with the appropriate element  $x$  as defined above.

**SchreierGraph(A, B)**

**UnlabelledSchreierGraph(A, B)**

Given a finite group  $A$  defined on the generating set  $X$  and a subgroup  $B$  of  $A$ , construct the Schreier coset graph  $S$  for  $A$  over  $B$ , relative to  $X$ . The graph  $S$  is defined as follows: The vertices correspond to the cosets of  $B$  in  $A$ , and two vertices  $u, v$  are adjacent in  $S$  if and only if there exists an element  $x$  in  $X$  such that  $u*x = v$ .

The graph is available in both a labelled and an unlabelled version.

**OrbitalGraph(P, u, T)**

Let  $P$  be a transitive permutation group acting on the set  $\Omega = \{1, \dots, n\}$ . Let  $u$  be an element of  $\Omega$  and let  $T = \{t_1, \dots, t_r\}$  be a subset of  $\Omega$ . This function constructs the underlying graph  $G$  of the digraph corresponding to the union of  $P$ -orbits containing the pairs  $(u, t_1), \dots, (u, t_r)$ . Thus, if  $T$  defines a self-paired orbit  $\Delta$  of the stabilizer in  $P$  of the point  $u$ , this function constructs the orbital graph associated with  $\Delta$ .

**ClosureGraph(P, G)**

Let  $P$  be a permutation group acting on the set  $\Omega = \{1, \dots, n\}$ . Let  $G$  be a graph (digraph) with vertices  $v_1, \dots, v_n$ . This function adds the minimum number of edges to  $G$  so as to produce a graph (digraph)  $H$  which is left invariant by the group  $P$ .

**PaleyGraph(q)**

The Paley graph of  $\mathbf{F}_q$  where  $q$  must be a prime power equivalent to 1 mod 4. Vertices are in bijection with elements of  $\mathbf{F}_q$  and distinct elements are adjacent when their difference is a square in the field.

**PaleyTournament(q)**

The Paley tournament of  $\mathbf{F}_q$  where  $q$  must be a prime power equivalent to 3 mod 4. Vertices are in bijection with elements of  $\mathbf{F}_q$  and there is an edge from  $u$  to  $v$  when  $u \neq v$  and  $v - u$  is a square in the field.

## 149.9.2 Graphs Constructed from Designs

**IncidenceGraph(D)**

Given an incidence structure  $D = (X, \mathcal{B})$ , construct the incidence graph  $G$  of  $D$ . The vertices of  $G$  is  $X \cup \mathcal{B}$ . The adjacency rules are as follows: No two vertices of  $X$  are adjacent; no two vertices of  $\mathcal{B}$  are adjacent; a vertex  $x \in X$  is adjacent to a vertex  $B \in \mathcal{B}$  if and only if  $x \in B$ .



**PointGraph(D)**

Given an incidence structure  $D = (X, \mathcal{B})$ , construct the point graph  $G$  of  $D$ . The vertex-set of  $G$  is  $X$ . Vertices  $x \in X$ ,  $y \in X$  are adjacent in  $G$  if there is a block  $B \in \mathcal{B}$  such that  $x \in B$  and  $y \in B$ .

**BlockGraph(D)**

The block graph of the incidence structure  $D$ ; i.e. the point graph of the dual of  $D$ .

**IncidenceGraph(P)**

Given a plane  $P$  with point-set  $V$  and line-set  $L$ , construct the incidence graph  $G$  of  $P$ . The vertex-set of  $G$  is  $V \cup L$ . The adjacency rules are as follows: No two vertices of  $V$  are adjacent; no two vertices of  $L$  are adjacent; a vertex  $v \in V$  is adjacent to a vertex  $a \in L$  if and only if  $v$  lies on  $a$ .

**PointGraph(P)**

Given a plane  $P$  with point-set  $V$  and line-set  $L$ , construct the point graph  $G$  of  $P$ . The vertex-set of  $G$  is  $V$ . Vertices  $u, v \in V$  are adjacent in  $G$  iff there is a line in  $L$  that contains them both.

**LineGraph(P)**

Given a plane  $P$  with point-set  $V$  and line-set  $L$ , construct the point graph  $G$  of  $P$ . The vertex-set of  $G$  is  $L$ . Lines  $a, b \in L$  are adjacent in  $G$  iff there is a vertex in  $V$  that lies on them both.

**HadamardGraph(H : parameters)****Labels**

BOOLELT

*Default : false*

The graph of the  $\pm 1$  matrix  $H$  as described in Brendan D. McKay's note "Hadamard equivalence via graph isomorphism" (with self-loops omitted). The parameter **Labels** is set to **false** by default, but when set to **true**, the vertices associated with rows are labelled "row" and the others "col". Those labelled "row" are those given loops in McKay's paper.

**149.9.3 Miscellaneous Graph Constructions****Converse(G)**

Returns the converse  $H$  of the directed graph  $G$ : if  $[u, v]$  is an edge of  $G$  then  $[v, u]$  is an edge of  $H$ .

**OddGraph(n)**

The  $n$ th odd graph. Vertices are  $(n - 1)$ -subsets of a  $(2n - 1)$ -set with vertices adjacent if and only if the  $(n - 1)$ -subsets are disjoint.

**TriangularGraph(*n*)**

The  $n$ th triangular graph. Vertices are 2-subsets of a  $n$ -set with vertices adjacent if and only if the 2-subsets are unequal and not disjoint.

**SquareLatticeGraph(*n*)**

The  $n$ th square lattice graph. This is the cartesian product of the  $n$ th complete graph with itself.

**ClebschGraph()**

**ShrikhandeGraph()**

**GewirtzGraph()**

Return the named graph.

**ChangGraphs()**

Return a sequence of the three Chang graphs.

## 149.10 Elementary Invariants of a Graph

**Order(*G*)**

**NumberOfVertices(*G*)**

The number of vertices of the graph  $G$ .

**Size(*G*)**

**NumberOfEdges(*G*)**

The number of edges of the graph  $G$ .

**CharacteristicPolynomial(*G*)**

The characteristic polynomial (over the integers) of the graph  $G$ ; i.e. the characteristic polynomial of the adjacency matrix of  $G$ .

**Spectrum(*G*)**

The spectrum of the graph  $G$ ; i.e. the roots of the characteristic polynomial of  $G$ . The roots are returned as a set of tuples, each containing a root and its multiplicity.

## 149.11 Elementary Graph Predicates

Let  $G$  and  $H$  be two graphs. For clarity, we list here the conditions under which  $G$  is equal to  $H$  and  $H$  is a subgraph of  $G$ . The conditions take into account the fact that the graphs may have a support and that their vertex and edges may be decorated.

The graphs  $G$  and  $H$  are equal if and only if:

- they are of the same type,
- they are structurally identical,
- they have the same support,
- they have identical vertex and edge labels,
- if applicable, the total capacity from  $u$  to  $v$  in  $G$  is equal to the total capacity from  $u$  to  $v$  in  $H$ .

Also,  $H$  is a subgraph of  $G$  if and only if:

- they are of the same type,
- $H$  is a structural subgraph of  $G$ ,
- any vertex  $v$  in  $H$  has the same support as the vertex  $\text{VertexSet}(G) ! v$  in  $G$ ,
- any vertex  $v$  in  $H$  has the same label as the vertex  $\text{VertexSet}(G) ! v$  in  $G$ ,
- any edge  $e$  in  $H$  has the same label as the edge  $\text{EdgeSet}(G) ! e$  in  $G$ ,
- if applicable, the total capacity from  $u$  to  $v$  in  $G$  is at least as large as the total capacity from  $u$  to  $v$  in  $H$ .

Note that the truth value of the above two tests is not dependent on the weights of the edges of the graphs, should these edges be weighted. One could argue that given this shortcoming, the tests should not be dependent on the capacities of the edges either. We welcome users' comments on this matter.

$u \text{ adj } v$

Let  $u$  and  $v$  be two vertices of the same graph  $G$ . If  $G$  is undirected, returns **true** if and only if  $u$  and  $v$  are adjacent. If  $G$  is directed, returns **true** if and only if there is an edge directed from  $u$  to  $v$ .

$e \text{ adj } f$

Let  $e$  and  $f$  be two edges of the same graph  $G$ . If  $G$  is undirected, returns **true** if and only if  $e$  and  $f$  share a common vertex. If  $G$  is directed, returns **true** if and only if the terminal vertex of  $e$  ( $f$ ) is the initial vertex of  $f$  ( $e$ ).

$u \text{ notadj } v$

The negation of the **adj** predicate applied to vertices.

$e \text{ notadj } f$

The negation of the **adj** predicate applied to edges.

u in e

Let  $u$  be a vertex and  $e$  an edge of a graph  $G$ . Returns **true** if and only if  $u$  is an end-vertex of  $e$ .

u notin e

The negation of the **in** predicate applied to a vertex with respect to an edge.

G eq H

Returns **true** if the graphs  $G$  and  $H$  are identical, otherwise **false**.  $G$  and  $H$  are identical if and only if:

- they are structurally identical,
- they have the same support,
- they have identical vertex and edge labels,
- if applicable, the total capacity from  $u$  to  $v$  in  $G$  is equal to the total capacity from  $u$  to  $v$  in  $H$ .

Thus, as an example, if  $G$  and  $H$  are structurally identical graphs, and the vertices of  $G$  are labelled while the vertices of  $H$  are not, then **G eq H** returns **false**.

IsSubgraph(G, H)

Returns **true** if  $H$  is a subgraph of  $G$ , otherwise **false**.  $H$  is a subgraph of  $G$  if and only if:

- $H$  can be determined to be a structural subgraph of  $G$ ,
- any vertex  $v$  in  $H$  has the same support as the vertex **VertexSet(G)!v** in  $G$ ,
- any vertex  $v$  in  $H$  has the same label as the vertex **VertexSet(G)!v** in  $G$ ,
- any edge  $e$  in  $H$  has the same label as the edge **EdgeSet(G)!e** in  $G$ .
- if applicable, the total capacity from  $u$  to  $v$  in  $G$  is at least as large as the total capacity from  $u$  to  $v$  in  $H$ .

IsBipartite(G)

Returns **true** if the graph  $G$  is a bipartite graph, otherwise **false**.

IsComplete(G)

Returns **true** if the graph  $G$  on  $n$  vertices is the complete graph on  $n$  vertices, otherwise **false**.

IsEulerian(G)

Returns **true** if the graph  $G$  is an eulerian graph, otherwise **false**. A eulerian graph is a graph whose vertices have all even degree. An eulerian digraph is a digraph whose vertices have same in and outdegree. That is, if  $D$  is a digraph,  $D$  is eulerian if and only if **OutDegree(v)** equals **InDegree(v)** for all vertices  $v$  of  $D$ .

**IsForest( $G$ )**

Returns **true** if the graph  $G$  is a forest, i.e. does not possess any cycles, otherwise **false**.

**IsEmpty( $G$ )**

Returns **true** if the graph  $G$  is an empty graph, otherwise **false**. A graph is empty if its edge-set  $E$  is empty.

**IsNull( $G$ )**

Returns **true** if the graph  $G$  is a null graph, otherwise **false**. A graph is null if its vertex-set  $V$  is empty.

**IsPath( $G$ )**

Returns **true** if the graph  $G$  is a path graph, otherwise **false**.

**IsPolygon( $G$ )**

Returns **true** if the graph  $G$  is a polygon graph, otherwise **false**.

**IsRegular( $G$ )**

Returns **true** if the graph  $G$  is a regular graph, otherwise **false**.

**IsTree( $G$ )**

Returns **true** if the graph  $G$  is a tree, otherwise **false**.

## 149.12 Adjacency and Degree

### 149.12.1 Adjacency and Degree Functions for a Graph

**Degree( $u$ )**

Given a vertex  $u$  of the graph  $G$ , return the degree of  $u$ , ie the number of edges incident to  $u$ .

**Alldeg( $G, n$ )**

Given a graph  $G$ , and a non-negative integer  $n$ , return the set of all vertices of  $G$  that have degree equal to  $n$ .

**MaximumDegree( $G$ )****Maxdeg( $G$ )**

The maximum of the degrees of the vertices of the graph  $G$ . This function returns two values: the maximum degree, and a vertex of  $G$  having that degree.

MinimumDegree( $G$ )

Mindeg( $G$ )

The minimum of the degrees of the vertices of the graph  $G$ . This function returns two values: the minimum degree, and a vertex of  $G$  having that degree.

DegreeSequence( $G$ )

Given a graph  $G$  such that the maximum degree of any vertex of  $G$  is  $r$ , return a sequence  $D$  of length  $r + 1$ , such that  $D[i]$ ,  $1 \leq i \leq r + 1$ , is the number of vertices in  $G$  having degree  $i - 1$ .

Valence( $G$ )

Given a regular graph  $G$ , return the valence of  $G$  (the degree of any vertex).

Neighbours( $u$ )

Neighbors( $u$ )

Given a vertex  $u$  of the graph  $G$ , return the set of vertices of  $G$  that are adjacent to  $u$ .

IncidentEdges( $u$ )

The set of all edges incident with the vertex  $u$ .

Bipartition( $G$ )

Given a bipartite graph  $G$ , return its two partite sets in the form of a pair of subsets of  $V(G)$ .

MinimumDominatingSet( $G$ )

A dominating set  $S$  of a graph  $G$  is such that the vertices of  $S$  together with the vertices adjacent to vertices in  $S$  form the vertex-set of  $G$ . A dominating set  $S$  is minimal if no proper subset of  $S$  is a dominating set. A minimum dominating set is a minimal dominating set of smallest size. The algorithm implemented is a backtrack algorithm (see [Chr75] p. 41).

### 149.12.2 Adjacency and Degree Functions for a Digraph

InDegree( $u$ )

The number of edges directed into the vertex  $u$  belonging to the directed graph  $G$ .

OutDegree( $u$ )

The number of edges of the form  $uv$  where  $u$  is a vertex belonging to the directed graph  $G$ .

Degree( $u$ )

Given a vertex  $u$  belonging to the digraph  $G$ , return the total degree of  $u$ , i.e. the sum of the in-degree and out-degree for  $u$ .

Alldeg( $G, n$ )

Given a digraph  $G$ , and a non-negative integer  $n$ , return the set of all vertices of  $G$  that have total degree equal to  $n$ .

MaximumInDegree( $G$ )

Maxindeg( $G$ )

The maximum indegree of the vertices of the digraph  $G$ . This function returns two values: the maximum indegree, and the first vertex of  $G$  having that degree.

MaximumOutDegree( $G$ )

Maxoutdeg( $G$ )

The maximum outdegree of the vertices of the digraph  $G$ . This function returns two values: the maximum outdegree, and the first vertex of  $G$  having that degree.

MinimumInDegree( $G$ )

Minindeg( $G$ )

The minimum indegree of the vertices of the digraph  $G$ . This function returns two values: the minimum indegree, and the first vertex of  $G$  having that degree.

MinimumOutDegree( $G$ )

Minoutdeg( $G$ )

The minimum outdegree of the vertices of the digraph  $G$ . This function returns two values: the minimum outdegree, and the first vertex of  $G$  having that degree.

MaximumDegree( $G$ )

Maxdeg( $G$ )

The maximum total degree of the vertices of the digraph  $G$ . This function returns two values: the maximum total degree, and the first vertex of  $G$  having that degree.

MinimumDegree( $G$ )

Mindeg( $G$ )

The minimum total degree of the vertices of the digraph  $G$ . This function returns two values: the minimum total degree, and the first vertex of  $G$  having that degree.

**DegreeSequence(*G*)**

Given a digraph  $G$  such that the maximum degree of any vertex of  $G$  is  $r$ , return a sequence  $D$  of length  $r + 1$ , such that  $D[i]$ ,  $1 \leq i \leq r + 1$ , is the number of vertices in  $G$  having degree  $i - 1$ .

**InNeighbours(*u*)****InNeighbors(*u*)**

Given a vertex  $u$  of the digraph  $G$ , return the set containing all vertices  $v$  such that  $vu$  is an edge in the digraph, i.e. the starting points of all edges that are directed into the vertex  $u$ .

**OutNeighbours(*u*)****OutNeighbors(*u*)**

Given a vertex  $u$  of the digraph  $G$ , return the set of vertices  $v$  of  $G$  such that  $uv$  is an edge in the graph  $G$ , i.e. the set of vertices  $v$  that are the end vertices of edges directed from  $u$  to  $v$ .

**IncidentEdges(*u*)**

The set of all edges incident with the vertex  $u$ .

## 149.13 Connectedness

### 149.13.1 Connectedness in a Graph

**IsConnected(*G*)**

Returns **true** if and only if the graph  $G$  is connected.

**Components(*G*)**

The connected components of the graph  $G$ . These are returned in the form of a sequence of subsets of the vertex-set of  $G$ .

**Component(*u*)**

The subgraph corresponding to the connected component of the graph  $G$  containing vertex  $u$ . The support and vertex/edge decorations are *not* retained in the resulting (structural) subgraph.

**IsSeparable(*G*)**

Returns **true** if and only if the graph  $G$  is connected and has at least one cut vertex.

**IsBiconnected(*G*)**

Returns **true** if and only if the graph  $G$  is biconnected.



**CutVertices( $G$ )**

The set of cut vertices for the connected graph  $G$  (a set of vertices).

**Bicomponents( $G$ )**

The biconnected components of the graph  $G$ . These are returned in the form of a sequence of subsets of the vertex-set of  $G$ . The graph may be disconnected.

**149.13.2 Connectedness in a Digraph****IsStronglyConnected( $G$ )**

Returns **true** if and only if the digraph  $G$  is strongly connected.

**IsWeaklyConnected( $G$ )**

Returns **true** if and only if the digraph  $G$  is weakly connected.

**StronglyConnectedComponents( $G$ )**

The strongly connected components of the digraph  $G$ . These are returned in the form of a sequence of subsets of the vertex-set of  $G$ .

**Component( $u$ )**

The subgraph corresponding to the connected component of the digraph  $G$  containing vertex  $u$ . The support and vertex/edge decorations are *not* retained in the resulting (structural) subgraph.

**149.13.3 Graph Triconnectivity**

The linear-time triconnectivity algorithm by Hopcroft and Tarjan [HT73] has been implemented with corrections of our own and from C. Gutwenger and P. Mutzel [GM01]. This algorithm requires that the graph has a sparse representation. The input graph may be disconnected or have cut vertices.

The algorithm completely splits the graph into components that can later be reconstructed as triconnected graphs: It is our belief that the word “tricomponents” is misused in all the papers discussing Hopcroft and Tarjan’s algorithm, including [HT73] and [GM01]. Let us clarify this point.

Assume that  $G$  is a biconnected but not triconnected graph with vertex-set  $V$  and with one separation pair  $(a, b)$  which splits  $V - \{a, b\}$  into  $V_1$  and  $V_2$ . That is, for any vertices  $u \in V_1$  and  $v \in V_2$ , every path from  $u$  to  $v$  in  $G$  contains at least one of  $a$  or  $b$ .

Now, the algorithm will correctly find the separation pair  $(a, b)$  and return the two subsets of  $V$ ,  $V_1 + \{a, b\}$  and  $V_2 + \{a, b\}$ . Let  $G_1$  and  $G_2$  be the subgraphs of  $G$  induced by  $V_1 + \{a, b\}$  and  $V_2 + \{a, b\}$  respectively. We say that the algorithm *splits*  $G$  into  $G_1$  and  $G_2$ .

But  $G_1$  and  $G_2$  are triconnected if and only if there is an edge  $(a, b)$  in  $G$ . We call  $G_1$  and  $G_2$  the *split components* of  $G$ .

More generally, if  $G_1, \dots, G_m$  are the split components of a graph  $G$ , one can make the  $G_i$  triconnected by adding the edges  $(a, b)$  to them for all the separation pairs  $(a, b)$  of  $G$  with  $a, b \in G_i$  for some  $i$ .

The algorithm finds all the cut vertices and/or the separation pairs of a graph, unless only the boolean test is applied. When this is the case the algorithm stops as soon as a cut vertex or separation pair is found.

The triconnectivity algorithm only applies to undirected graphs.

**IsTriconnected(G)**

Returns **true** if and only if the graph  $G$  is triconnected.

**Splitcomponents(G)**

The split components of the graph  $G$ . These are returned in the form of a sequence of subsets of the vertex-set of  $G$ . The graph may be disconnected. The second return value returns the cut vertices and the separation pairs.

**SeparationVertices(G)**

The cut vertices and/or the separation pairs of the graph  $G$  as a sequence of vertices or pairs of vertices of  $G$ . The graph may be disconnected. The second return value returns  $G$ 's split components.

#### Example H149E12

---

```
> G := Graph< 11 |
> {1, 3}, {1, 4}, {1, 7}, {1, 11}, {1, 2},
> {2, 4}, {2, 3},
> {3, 4},
> {4, 7}, {4, 11}, {4, 7}, {4, 5},
> {5, 10}, {5, 9}, {5, 8}, {5, 6},
> {6, 8}, {6, 7},
> {7, 8},
> {9, 11}, {9, 10},
> {10, 11}: SparseRep := true >;
> TS, PS := Splitcomponents(G);
> TS;
[
  { 5, 6, 7, 8 },
  { 5, 9, 10, 11 },
  { 1, 4, 5, 7, 11 },
  { 1, 4 },
  { 1, 2, 3, 4 }
]
> PS;
[
  [ 5, 7 ],
  [ 5, 11 ],
```

```
[ 1, 4 ]
]
```

Let  $G_1$  be the subgraph induced by  $TS[1]$ . We will verify that it is not triconnected and we make it triconnected by adding the edge  $[5, 7]$ , since  $[5, 7]$  is a separation pair of  $G$  whose vertices belong to  $G_1$ .

```
> G1 := sub< G | TS[1] >;
> IsTriconnected(G1);
false
> AddEdge(~G1, Index(VertexSet(G1)!VertexSet(G)!5),
>         Index(VertexSet(G1)!VertexSet(G)!7));
> IsTriconnected(G1);
true
```

### 149.13.4 Maximum Matching in Bipartite Graphs

The maximum matching algorithm is a flow-based algorithm. We have implemented two maximum flow algorithms, the Dinic algorithm, and a push-relabel algorithm. The latter is almost always the most efficient, it may be outperformed by Dinic only in the case of very sparse graphs. For a full discussion of these algorithms, the reader is referred to Section 151.4 in Chapter 150. Example H151E4 in this same chapter actually is an implementation in MAGMA code of the maximum matching algorithm.

MaximumMatching(G)
--------------------

**A1**

MONSTGELT

*Default : "PushRelabel"*

A maximum matching in the bipartite graph  $G$ . The matching is returned as a sequence of edges of  $G$ . The parameter **A1** enables the user to select the algorithm which is to be used: **A1** := "PushRelabel" or **A1** := "Dinic".

#### Example H149E13

We begin by creating a random bipartite graph. The matching algorithm is flow-based and is one of the algorithms which deals with graphs as adjacency lists. For this reason we might as well create the graph with a sparse representation (see Section 149.3).

```
> n := 5;
> m := 7;
> d := 3;
> G := EmptyGraph(n);
> H := EmptyGraph(m);
> G := G join H;
> for u in [1..n] do
>   t := Random(0, d);
>   for tt in [1..t] do
>     v := Random(n+1, n+m);
```

```

>      AddEdge(~G, u, v);
>    end for;
> end for;
> G := Graph< Order(G) | G : SparseRep := true >;
>
> IsBipartite(G);
true
> Bipartition(G);
[
  { 1, 12, 2, 3, 4, 5, 7 },
  { 11, 6, 8, 9, 10 }
]
> MaximumMatching(G);
[ {1, 11}, {4, 10}, {2, 9}, {5, 8}, {3, 6} ]

```

---

### 149.13.5 General Vertex and Edge Connectivity in Graphs and Digraphs

As for the maximum matching algorithm (Subsection 149.13.4) the vertex and edge connectivity algorithms are flow-based algorithms. We have implemented two flow algorithms, Dinic and a push relabel method. In general Dinic outperforms the push relabel method only in the case of very sparse graphs. This is particularly visible when computing the edge connectivity of a graph. For full details on those flow algorithms, refer to Section 151.4 in Chapter 150. For details on how vertex and edge connectivity are implemented, see [Eve79].

The general connectivity algorithms apply to both undirected and directed graphs.

#### VertexSeparator(G)

**A1**

MONSTGEALT

*Default : "PushRelabel"*

If  $G$  is an undirected graph, a *vertex separator* for  $G$  is a smallest set of vertices  $S$  such that for any  $u, v \in V(G)$ , every path connecting  $u$  and  $v$  passes through at least one vertex of  $S$ .

If  $G$  is a directed graph, a vertex separator for  $G$  is a smallest set of vertices  $S$  such that for any  $u, v \in V(G)$ , every directed path from  $u$  to  $v$  passes through at least one vertex of  $S$ .

**VertexSeparator** returns the vertex separator of  $G$  as a sequence of vertices of  $G$ . The parameter **A1** enables the user to select the algorithm which is to be used: **A1** := "PushRelabel" or **A1** := "Dinic".

#### VertexConnectivity(G)

**A1**

MONSTGEALT

*Default : "PushRelabel"*

Returns the vertex connectivity of the graph  $G$ , the size of a minimum vertex separator of  $G$ . Also returns a vertex separator for  $G$  as the second value. The parameter **A1** enables the user to select the algorithm which is to be used: **A1** := "PushRelabel" or **A1** := "Dinic".

**IsKVertexConnected(G, k)**

A1

MONSTGELT

*Default : "PushRelabel"*

Returns **true** if the graph  $G$ 's vertex connectivity is at least  $k$ , **false** otherwise. The parameter A1 enables the user to select the algorithm which is to be used: A1 := "PushRelabel" or A1 := "Dinic".

**EdgeSeparator(G)**

A1

MONSTGELT

*Default : "PushRelabel"*

If  $G$  is an undirected graph, an *edge separator* for  $G$  is a smallest set of edges  $T$  such that for any  $u, v \in V(G)$ , every path connecting  $u$  and  $v$  passes through at least one edge of  $T$ .

If  $G$  is a directed graph, an edge separator for  $G$  is a smallest set of edges  $T$  such that for any  $u, v \in V(G)$ , every directed path from  $u$  to  $v$  passes through at least one edge of  $T$ .

EdgeSeparator returns the edge separator of  $G$  as a sequence of edges of  $G$ . The parameter A1 enables the user to select the algorithm which is to be used: A1 := "PushRelabel" or A1 := "Dinic".

**EdgeConnectivity(G)**

A1

MONSTGELT

*Default : "PushRelabel"*

Returns the edge connectivity of the graph  $G$ , the size of a minimum edge separator of  $G$ . Also returns as the second value an edge separator for  $G$ . The parameter A1 enables the user to select the algorithm which is to be used: A1 := "PushRelabel" or A1 := "Dinic".

**IsKEdgeConnected(G, k)**

A1

MONSTGELT

*Default : "PushRelabel"*

Returns **true** if the graph  $G$ 's edge connectivity is at least  $k$ , **false** otherwise. The parameter A1 enables the user to select the algorithm which is to be used: A1 := "PushRelabel" or A1 := "Dinic".

**Example H149E14**

We compute the vertex and edge connectivity for a small undirected graph and we perform a few checks. The connectivity algorithm is one of the algorithms which deals with graphs as adjacency lists. For this reason we might as well create the graph with a sparse representation (see Section 149.3).

```
> G := Graph< 4 | {1, 2}, {1, 3}, {2, 3}, {2, 4}, {3, 4}
> : SparseRep := true >;
> IsConnected(G);
true
>
> vc := VertexConnectivity(G);
> S := VertexSeparator(G);
```

```

> vc;
2
> S;
[ 3, 2 ]
>
> H := G;
> vs := [ Integers() | Index(x) : x in S ];
> RemoveVertices(~H, vs);
> IsConnected(H);
false
> for k in [0..vc] do
>   IsKVertexConnected(G, k);
> end for;
true
true
true
> for k in [vc+1..Order(G)-1] do
>   IsKVertexConnected(G, k);
> end for;
false
>
>
>
> ec := EdgeConnectivity(G);
> T := EdgeSeparator(G);
> ec;
2
> T;
[ {1, 2}, {1, 3} ]
>
> H := G;
> M := [ { Index(e[1]), Index(e[2]) }
> where e := SetToSequence(EndVertices(e)) : e in T ];
> RemoveEdges(~H, M);
> IsConnected(H);
false
> for k in [0..ec] do
>   IsKEdgeConnected(G, k);
> end for;
true
true
true
> for k in [ec+1..Order(G)-1] do
>   IsKEdgeConnected(G, k);
> end for;
false

```

---

## 149.14 Distances, Paths and Circuits in a Graph

The distance functions apply to graphs as well as to digraphs.

### 149.14.1 Distances, Paths and Circuits in a Possibly Weighted Graph

There are a few distance and path functions that take into account the fact that the edges of the graph under consideration may be weighted. If the graph is not weighted, then the distance between two vertices is taken to be the length of the shortest path between the vertices. For more details on distance and paths functions in weighted graphs, refer to Section 150.12 in Chapter 150.

**Reachable( $u$ ,  $v$ )**

Return **true** if and only if vertices  $u$  and  $v$  of a graph  $G$  are in the same component of  $G$ .

**Distance( $u$ ,  $v$ )**

Given two vertices  $u$  and  $v$  belonging to a graph  $G$ , return the length of a shortest path from  $u$  to  $v$ . If the graph is weighted, the function returns the total weight of the path from  $u$  to  $v$ . Results in an error if there is no path in  $G$  from  $u$  to  $v$ .

**Geodesic( $u$ ,  $v$ )**

Given vertices  $u$  and  $v$  belonging to the graph  $G$ , return a sequence of vertices  $u = v_1, v_2, \dots, v_n = v$  such that the sequence of edges  $v_1v_2, v_2v_3, \dots, v_{n-1}v_n$  forms a shortest path joining  $u$  and  $v$ . Results in an error if there is no path in  $G$  from  $u$  to  $v$ .

### 149.14.2 Distances, Paths and Circuits in a Non-Weighted Graph

All the distance and path functions listed below do not take account of the fact that the graph edges may be weighted. *The distance between two vertices is understood in its usual meaning of being the length of the shortest path between the vertices.*

**Diameter( $G$ )**

The length of the longest path in the graph  $G$ . If  $G$  is not connected, the value  $-1$  is returned. To see how the automorphism group of  $G$  is computed see Subsection 149.22.3.

**DiameterPath( $G$ )**

A sequence of vertices defining a longest path if the graph  $G$  is connected, or the empty sequence if  $G$  is not connected. To see how the automorphism group of  $G$  is computed see Subsection 149.22.3.

**Ball(*u*, *n*)**

Given a vertex  $u$  belonging to a graph  $G$ , and a non-negative integer  $n$ , return the set of vertices of  $G$  that are at distance  $n$  or less from  $u$ .

**Sphere(*u*, *n*)**

Given a vertex  $u$  belonging to the graph  $G$ , and a non-negative integer  $n$ , return the set of vertices of  $G$  that are at distance  $n$  from  $u$ .

**DistancePartition(*u*)**

Given a vertex  $u$  belonging to the graph  $G$ , return the partition  $P_0 \cup P_1 \cup \dots \cup P_d$  of the vertex-set of  $G$ , where  $P_i$  is the set of vertices lying at distance  $i$  from vertex  $u$ . The partition is returned as a sequence of sets. Any vertices not connected to  $u$  are treated as being at infinite distance from  $u$  and are returned as the last set of the partition.

**IsEquitable(*G*, *P*)**

Given a partition  $P$  of the vertex-set  $V(G)$  of the graph  $G$ , return the value **true** if  $P$  is an equitable partition.

**EquitablePartition(*P*, *G*)**

Given a partition  $P$  of the vertex-set  $V(G)$  of the graph  $G$ , return the coarsest partition of  $V(G)$  that refines  $P$  and which is equitable.

**Girth(*G*)**

The girth of graph  $G$ , i.e. the length of a shortest cycle. To see how the automorphism group of  $G$  is computed see Subsection 149.22.3.

**GirthCycle(*G*)**

A cycle of shortest length in the graph  $G$ . To see how the automorphism group of  $G$  is computed see Subsection 149.22.3.

## 149.15 Maximum Flow, Minimum Cut, and Shortest Paths

Whenever the edges of a graph are assigned a capacity it is possible to compute the maximum flow and the minimum cut of the graph. Whenever the edges of a graph are assigned a weight it is possible to find shortest paths in the graph. If one wants to perform these computations on a graph where none of its edges have been assigned a capacity or a weight, then the edges are taken to have a capacity or weight of one, and the computations are carried out accordingly.

The flow-based and shortest-paths algorithms and the resulting functionalities are fully documented in Chapter 150, Sections 151.4 and 150.12 respectively.



## 149.16 Matrices and Vector Spaces Associated with a Graph or Digraph

### AdjacencyMatrix(G)

Returns the adjacency matrix for the  $(p, q)$  graph  $G$  as an element of the matrix ring  $M_p(Z)$ .

### DistanceMatrix(G)

Returns the distance matrix  $A$  for the  $(p, q)$  graph  $G$  as an element of the matrix ring  $M_p(Z)$ . The  $(i, j)$ -th entry of  $A$  gives the distance between vertices  $v_i$  and  $v_j$  of  $G$ .

### IncidenceMatrix(G)

Returns the incidence matrix  $M$  for the  $(p, q)$  graph  $G$  as an element of the matrix bimodule  $M^{p \times q}(Z)$ .

If  $G$  is a graph, then entry  $(i, j)$  of  $M$  is 1 if the vertex  $v_i$  of  $G$  lies on the edge  $e_j$  of  $G$ . Otherwise entry  $(i, j)$  is zero.

If  $G$  is a digraph, entry  $(i, j)$  of  $M$  is 1 if vertex  $v_i$  is the initial vertex of the edge  $e_j$ , and  $-1$  if  $v_i$  is the final vertex of the edge  $e_j$ . Otherwise entry  $(i, j)$  is zero. If  $e_j$  is a loop, then entry  $(i, j)$  may be either 1 or  $-1$ .

### IntersectionMatrix(G, P)

Given an ordered equitable partition  $P = P_1 \cup P_2 \cup \dots \cup P_r$  of the vertex-set of the graph  $G$ , return the intersection matrix  $T$  for the partition. Thus, entry  $T[i, j]$  is the number of vertices of the set  $P_j$  that are adjacent to a vertex of the set  $P_i$ .

## 149.17 Spanning Trees of a Graph or Digraph

### SpanningTree(G)

Given a connected (undirected) graph  $G$ , construct a spanning tree for  $G$  rooted at an arbitrary vertex of  $G$ . The spanning tree is returned as a subgraph of  $G$ . The support and vertex/edge decorations are *not* retained in the resulting (structural) subgraph.

### SpanningForest(G)

Given a graph  $G$ , construct a spanning forest for  $G$ . The forest is returned as a subgraph of  $G$ . The support and vertex/edge decorations are *not* retained in the resulting (structural) subgraph.

BreadthFirstSearchTree(u)
---------------------------

BFSTree(u)
------------

Given a vertex  $u$  belonging to the graph  $G$ , return a breadth-first search for  $G$  rooted at the vertex  $u$ . The tree is returned as a subgraph of  $G$ . The support and vertex/edge decorations are *not* retained in the resulting (structural) subgraph. Note that  $G$  may be disconnected.

DepthFirstSearchTree(u)
-------------------------

DFSTree(u)
------------

Given a vertex  $u$  belonging to the graph  $G$ , return a depth-first search tree  $T$  for  $G$  rooted at the vertex  $u$ . The tree  $T$  is returned as a subgraph of  $G$ . The support and vertex/edge decorations are *not* retained in the resulting (structural) subgraph. Note that  $G$  may be disconnected.

The fourth return argument returns, for each vertex  $u$  of  $G$ , the tree order of  $u$ , that is, the order in which the vertex  $u$  has been visited while performing the depth-first search. If  $T$  does not span  $G$  then the vertices of  $G$  not in  $T$  are given tree order from  $\text{Order}(T) + 1$  to  $\text{Order}(G)$ .

## 149.18 Directed Trees

In the following functions, the graph is assumed to be a directed tree. This means it is a tree containing a root vertex and all edges are directed away from that vertex.

IsRootedTree(G)
-----------------

Returns **true** exactly when the directed graph  $G$  is a tree having a vertex  $v$  such that all edges are directed away from  $v$ . In this case, the root vertex  $v$  is returned as a second value.

Root(G)
---------

The root vertex of a rooted tree.

IsRoot(v)
-----------

Returns **true** if and only if the graph containing the vertex  $v$  is directed as a rooted tree with  $v$  as root.

RootSide(v)
-------------

When the graph containing the vertex  $v$  is directed as a rooted tree, this returns the unique neighbouring vertex to  $v$  which is closer to the root vertex. If  $v$  is the root vertex, it is returned itself.

**VertexPath( $u, v$ )**

A sequence of vertices comprising a path in a directed graph from the vertex  $u$  to the vertex  $v$ . The path does not necessarily respect the edge directions. Indeed it will first trace back to a common ancestor of  $u$  and  $v$  and then follow edge directions to  $v$ .

**BranchVertexPath( $u, v$ )**

The sequence of vertices on the vertex path from the vertex  $u$  to the vertex  $v$  having valency at least 3.

## 149.19 Colourings

The functions given here are not applicable to digraphs.

**ChromaticNumber( $G$ )**

The chromatic number of the graph  $G$ , that is, the minimum number of colours required to colour the vertices of  $G$  such that no two adjacent vertices share the same colour.

**OptimalVertexColouring( $G$ )**

An optimal vertex colouring of the graph  $G$  as a sequence of sets of vertices of  $G$ . Each set in the sequence contains the vertices which are given the same colour. Some optimal vertex colouring is returned, others may exist.

**ChromaticIndex( $G$ )**

The chromatic index of the graph  $G$ , that is, the minimum number of colours required to colour the edges of  $G$  such that no two adjacent edges share the same colour.

**OptimalEdgeColouring( $G$ )**

An optimal edge colouring of the graph  $G$  as a sequence of sets of edges of  $G$ . Each set in the sequence contains the edges which are given the same colour. Some optimal edge colouring is returned, others may exist.

**ChromaticPolynomial( $G$ )**

The chromatic polynomial of the graph  $G$ . For a given natural number  $x$ , the chromatic polynomial  $p_G(x)$  gives the number of colourings of  $G$  with colours  $1, 2, \dots, x$ .

**Example H149E15**

---

We illustrate the use of these functions with the following graph:

```
> G:=Graph< 5 | [{2,5}, {1,3,5}, {2,4,5}, {3,5}, {1,2,3,4} ]>;
> ChromaticNumber(G);
3
> OptimalVertexColouring(G);
[
  { 1, 3 },
  { 2, 4 },
  { 5 }
]
> ChromaticIndex(G);
4
> OptimalEdgeColouring(G);
[
  { {1, 2}, {3, 5} },
  { {2, 3}, {1, 5} },
  { {3, 4}, {2, 5} },
  { {4, 5} }
]
> ChromaticPolynomial(G);
$.1^5 - 7*$.1^4 + 18*$.1^3 - 20*$.1^2 + 8*$.1
```

---

## 149.20 Cliques, Independent Sets

The functions given here are not applicable to digraphs.

A *clique* of a graph  $G$  is a complete subgraph of  $G$ . A *maximal clique* is a clique which is not contained in any other clique. A *maximum clique* is a maximal clique of largest size.

An *independent set* of  $G$  is an empty subgraph of  $G$ . A *maximal (maximum) independent set* is defined in the same way as a maximal (maximum) clique. Note that an independent set of size  $k$  in the graph  $G$  is a clique of size  $k$  in the complement graph of  $G$ .

The clique and independent set functions presented below are implemented using one of two algorithms, called here *BranchAndBound* and *Dynamic*. The algorithms *BranchAndBound* and *Dynamic* are briefly described here.

*BranchAndBound* : The algorithm is an implementation of the branch and bound algorithm by Bron and Kerbosch [BK73].

The algorithm is designed to find maximal cliques and it has been adapted here to also find cliques of specific size which may not be maximal. It attempts to build a maximal clique by trying to expend the current maximal clique. Some heuristics are built into the algorithm which enable to prune the search tree.

*Dynamic* : The algorithm finds a clique of size exactly  $k$ , not necessarily maximal, in the graph  $G$  by using recursion and dynamic programming. If a clique of size  $k$  exists in  $G$ ,

then, for a given vertex  $v$  of  $G$ , there is either a clique of size  $k-1$  in the subgraph induced by the neighbours of  $v$ , or there is a clique of size  $k$  in the graph  $G-v$ . This is the recursive step. The *Dynamic* algorithm applies a different strategy (sorting of vertices and selection of vertices to consider) according to the order and density of the subgraph considered at the current level of recursion. This is achieved by dynamic programming, hence the name. This algorithm is due to Wendy Myrvold [WM].

**Comment:** When comparing both algorithms in the situation where the problem is to find a maximum clique one observes that in general *BranchAndBound* does better. However *Dynamic* outperforms *BranchAndBound* when the graphs under consideration are large (more than 400 vertices) random graphs with high density (larger than 0.5%). So far, it can only be said that the comparative behaviour of both algorithms is highly dependent on the structure of the graphs.

HasClique( $G, k$ )

Returns **true** if and only if the graph  $G$  has a maximal clique of size  $k$ . If **true**, returns such a clique as the second argument.

HasClique( $G, k, m : parameters$ )

A1

MONSTGELT

Default : “*BranchAndBound*”

If  $m$  is **true**, the function is **true** if and only if the graph  $G$  has a maximal clique of size  $k$ . If  $m$  is **false**, the function is **true** if and only if the graph  $G$  has a — not necessarily maximal — clique of size  $k$ . If the function is **true**, an appropriate clique is returned as the second argument. When  $m$  is **false**, the parameter A1 enables the user to select the algorithm which is to be used. When  $m$  is **true**, the parameter A1 is ignored.

The parameter A1 enables the user to select the algorithm which is to be used: A1 := “BranchAndBound” or A1 := “Dynamic”. See the description given at the beginning of this section.

The default is “BranchAndBound”.

HasClique( $G, k, m, f : parameters$ )

A1

MONSTGELT

Default : “*BranchAndBound*”

If  $m$  is **true** and  $f = 0$ , the function is **true** if and only if the graph  $G$  has a maximal clique of size  $k$ . If  $m$  is **true** and  $f = 1$ , the function is **true** if and only if the graph  $G$  has a maximal clique of size larger than or equal to  $k$ . If  $m$  is **true** and  $f = -1$ , the function is **true** if and only if the graph  $G$  has a maximal clique of size smaller than or equal to  $k$ . If  $m$  is **false**, the function is **true** if and only if the graph  $G$  has a — not necessarily maximal — clique of size  $k$ . If the function is **true**, an appropriate clique is returned as the second argument. When  $m$  is **false**, the parameter A1 enables the user to select the algorithm which is to be used. When  $m$  is **true** the parameter A1 is ignored, and when  $m$  is **false**, the flag  $f$  is ignored.

The parameter A1 enables the user to select the algorithm which is to be used: A1 := “BranchAndBound” or A1 := “Dynamic”. See the description given at the beginning of this section.

The default is "BranchAndBound".

MaximumClique( <i>G</i> : <i>parameters</i> )
---

A1

MONSTGELT

Default : "BranchAndBound"

Finds a maximum clique in the graph  $G$ . The clique is returned as a set of vertices.

The parameter A1 enables the user to select the algorithm which is to be used.

A1 := "BranchAndBound": See the description at the beginning of this section.

A1 := "Dynamic": Two steps are required here.

Step 1      Finding a lowerbound on the size of a maximum clique. This is achieved by using the *dsatur* colouring (*dsatur* stands for saturation degree) due to Brélaz [Bre79]. The *dsatur* colouring gives a reasonably good approximation to the size of a maximum clique, usually with a penalty of 2 to 3.

Step 2      Assume that the lowerbound found in Step 1 is  $l$ . Then a maximum clique is found by finding the largest possible clique of size  $k \geq l$  using the Dynamic algorithm.

The default is "BranchAndBound".

CliqueNumber( <i>G</i> : <i>parameters</i> )
--

A1

MONSTGELT

Default : "BranchAndBound"

Finds the size of a maximum clique in the graph  $G$ . The parameter A1 enables the user to select the algorithm which is to be used. A1 := "BranchAndBound": See the description given at the beginning of this section.

A1 := "Dynamic": See the function MaximumClique.

The default is "BranchAndBound".

AllCliques( <i>G</i> : <i>parameters</i> )
--

Limit

RNGINTELT

Default : 0

Returns all maximal cliques of the graph  $G$  as a sequence of sets of vertices. If Limit is set to a positive integer, returns Limit maximal cliques of  $G$ .

AllCliques( <i>G</i> , <i>k</i> : <i>parameters</i> )
---

Limit

RNGINTELT

Default : 0

Returns all maximal cliques of size  $k$  of the graph  $G$  as a sequence of sets of vertices. If Limit is set to a positive integer, returns Limit maximal cliques of size  $k$  of  $G$ .

<b>AllCliques</b> ( <i>G</i> , <i>k</i> , <i>m</i> : <i>parameters</i> )
--

<b>Limit</b>	RNGINTELT
--------------	-----------

Default : 0

<b>A1</b>	MONSTGELT
-----------	-----------

Default : "BranchAndBound"

If *m* is **true**, returns all maximal cliques of size *k* in the graph *G*. If *m* is **false**, returns all — not necessarily maximal — cliques of size *k*. When *m* is **false**, the parameter **A1** enables the user to select the algorithm which is to be used. When *m* is **true**, the parameter **A1** is ignored.

The parameter **A1** enables the user to select the algorithm which is to be used: **A1** := "BranchAndBound" or **A1** := "Dynamic". See the description given at the beginning of this section. The default is "BranchAndBound".

Except in the case where *m* is **false** and **A1** is "Dynamic", the parameter **Limit**, if set to a positive integer, limits the number of cliques returned.

Maximal independent sets or independent sets of a given size *k* in a graph *G* can be easily found by finding maximal cliques or cliques of size *k* in the complement of *G*. Only two functions which are concerned with independent sets are provided: one finds a maximum independent set and the other returns the independence number of a graph.

<b>MaximumIndependentSet</b> ( <i>G</i> : <i>parameters</i> )
---

<b>A1</b>	MONSTGELT
-----------	-----------

Default : "BranchAndBound"

Finds a maximum independent set in the graph *G*. The maximum independent set is returned as a set of vertices.

The parameter **A1** enables the user to select the algorithm which is to be used: **A1** := "BranchAndBound" or **A1** := "Dynamic". See the function **MaximumClique**. The default is "BranchAndBound".

<b>IndependenceNumber</b> ( <i>G</i> : <i>parameters</i> )
--

<b>A1</b>	MONSTGELT
-----------	-----------

Default : "BranchAndBound"

Finds the size of a maximum independent set in the graph *G*.

The parameter **A1** enables the user to select the algorithm which is to be used: **A1** := "BranchAndBound" or **A1** := "Dynamic". See the function **CliqueNumber**. The default is "BranchAndBound".

### Example H149E16

---

We illustrate the use of the clique functions with the following graph:

```
> G := Graph< 9 | [ {4,5,6,7,8,9}, {4,5,6,7,8,9}, {4,5,6,7,8,9},
>                   {1,2,3,7,8,9}, {1,2,3,7,8,9}, {1,2,3,7,8,9},
>                   {1,2,3,4,5,6}, {1,2,3,4,5,6}, {1,2,3,4,5,6} ]>;
> HasClique(G, 2);
false
> HasClique(G, 2, true);
false
> HasClique(G, 2, false);
```

```
true { 1, 4 }
> HasClique(G, 2, true: A1 := "Dynamic");
false
> HasClique(G, 2, false: A1 := "Dynamic");
true { 1, 9 }
> HasClique(G, 2, true, 1);
true { 1, 4, 7 }
> MaximumClique(G);
{ 1, 4, 7 }
> AC := AllCliques(G);
> #AC;
27
> AC3 := AllCliques(G,3);
> #AC3;
27
> AC eq AC3;
true
> AllCliques(G, 2, true);
[]
> AllCliques(G, 2, false);
[
  { 1, 4 },
  { 1, 5 },
  { 1, 6 },
  { 1, 7 },
  { 1, 8 },
  { 1, 9 },
  { 2, 4 },
  { 2, 5 },
  { 2, 6 },
  { 2, 7 },
  { 2, 8 },
  { 2, 9 },
  { 3, 4 },
  { 3, 5 },
  { 3, 6 },
  { 3, 7 },
  { 3, 8 },
  { 3, 9 },
  { 4, 7 },
  { 4, 8 },
  { 4, 9 },
  { 5, 7 },
  { 5, 8 },
  { 5, 9 },
  { 6, 7 },
  { 6, 8 },
  { 6, 9 }
```



]

## 149.21 Planar Graphs

A linear-time algorithm due to Boyer and Myrvold [BM01] has been implemented to test if a graph is planar, and to identify a Kuratowski subgraph if the graph is non-planar. This algorithm requires that the graph has a sparse representation.

The interest of this new algorithm is that it is very easy to implement when compared to previous linear-time planarity testers. In addition, the algorithm also isolates a Kuratowski subgraph, if any, in linear-time.

The concept underlying the tester is deceptively simple: the idea is to perform a depth first search of the graph and then to try to embed (by post-order traversal of the tree) the back edges from a vertex  $v$  to its descendants in such a way that they do not cross and such that vertices with connections to ancestors of  $v$  remain along the external face.

To decide the order in which to embed back edges from  $v$  to its descendants, the external faces of components rooted by children of  $v$  are explored top-down, avoiding paths containing vertices with ancestor connections. To decide whether a vertex  $u$  has ancestor connections and must therefore be kept on the external face, one only needs the least ancestor directly adjacent to  $u$  and the lowpoint information for the children of  $u$  that are separable from the parent of  $u$  in the partial embedding.

Each back edge  $[w, v]$  is added as soon as the descendant endpoint  $w$  is encountered. If  $w$  is not in the same biconnected component as  $v$ , then the biconnected components encountered during the traversal are merged at their respective cut vertices since the new edge  $[w, v]$  biconnects them. The merge is performed such that vertices with ancestor connections remain on the external face of the newly formed biconnected component.

Traversal is terminated when all back edges from  $v$  to its descendants are added or when the external face traversal is prevented from reaching the descendant endpoint of a back edge. This occurs when both external face paths extending from the least numbered vertex in a biconnected component contain vertices with ancestor connections that block further traversal to a vertex that is the descendant endpoint  $w$  of a back edge  $[w, v]$ . In this case, a Kuratowski subgraph is isolated based primarily on ancestor connections, external face paths and depth first search tree paths.

MAGMA is deeply grateful for the valuable help and collaboration from the authors of [BM01], in particular John Boyer.

**IsPlanar( $G$ )**

Tests whether the (undirected) graph  $G$  is planar. The graph may be disconnected. If the graph is non-planar then a Kuratowski subgraph of  $G$  is returned: That is, a subgraph of  $G$  homeomorphic to  $K_5$  or  $K_{3,3}$ . The support and vertex/edge decorations of  $G$  are *not* retained in this (structural) subgraph.

**Obstruction(G)**

Returns a Kuratowski obstruction if the graph is non-planar, or the empty graph if the graph is planar. The Kuratowski is a (structural) subgraph of  $G$ ; the support and vertex/edge decorations of  $G$  are not transferred to it.

**IsHomeomorphic(G : parameters)****Graph**

MONSTGELE

*Default :*

Tests if a graph is homeomorphic to either  $K_5$  or  $K_{3,3}$ . The parameter **Graph** must be set to either “K5” or “K33”; it has no default setting.

**Faces(G)**

Returns the faces of the planar graph  $G$  as sequences of the edges bordering the faces of  $G$ . If  $G$  is disconnected, then the face defined by an isolated vertex  $v$  is given as  $[v]$ .

**Face(u, v)**

Returns the face of the planar graph  $G$  bordered by the directed edge  $[u, v]$  as an ordered list of edges of  $G$ .

Note that a directed edge and an orientation determine a face uniquely: We can assume without loss of generality that the plane is given a clockwise orientation. Then given a directed edge  $e = [u_1, v_1]$ , the face defined by  $e$  is the ordered set of edges  $[u_1, v_1], [u_2, v_2], \dots, [u_m, v_m]$  such that  $v_i = u_{i+1}$  for all  $i$ ,  $1 \leq i < m$ ,  $v_m = u_1$ , and for each  $v_i = u_{i+1}$ , the neighbours of  $v_i$ ,  $u_i$  and  $v_{i+1}$ , are *consecutive* vertices in  $v_i$ 's adjacency list whose order is *anti-clockwise*.

**Face(e)**

Let  $e$  be the edge  $u, v$  of the planar graph  $G$  (recall that  $G$  is undirected). Then **Face(u, v)** returns the face bordered by the directed edge  $[u, v]$  as a sequence of edges of  $G$ .

**NFaces(G)****NumberOfFaces(G)**

Returns the number of faces of the planar graph  $G$ . In the case of a disconnected graph, an isolated vertex counts for one face.

**Embedding(G)**

Returns the graph  $G$ 's planar embedding as a sequence  $S$  where  $S[i]$  is a sequence of edges incident from vertex  $i$ .

**Embedding(v)**

Returns the ordered list of edges (in clockwise order say) incident from vertex  $v$ .

**PlanarDual(G)**

Constructs the dual  $G'$  of a planar graph  $G$ . The numbering of the vertices of  $G'$  corresponds to the order of the faces as returned by **Faces(G)**.

**Example H149E17**

---

We start with a small disconnected planar graph  $G$ : Notice how one of the faces of  $G$  is defined by the isolated vertex 4.

```
> G := Graph< 5 | [ {2, 3, 5}, {1, 5}, {1}, {}, {1, 2} ] >;
> G;
Graph
Vertex Neighbours
1      2 3 5 ;
2      1 5 ;
3      1 ;
4      ;
5      1 2 ;
> IsConnected(G);
false
> IsPlanar(G);
true
> Faces(G);
[
  [ {1, 2}, {2, 5}, {5, 1} ],
  [ {1, 5}, {5, 2}, {2, 1}, {1, 3}, {3, 1} ],
  [ 4 ]
]
> Embedding(G);
[
  [ {1, 2}, {1, 5}, {1, 3} ],
  [ {2, 5}, {2, 1} ],
  [ {3, 1} ],
  [],
  [ {5, 1}, {5, 2} ]
]
> Embedding(VertexSet(G)!1);
[ {1, 2}, {1, 5}, {1, 3} ]
```

Now let's turn to a simple non-planar graph whose obstruction is  $K_{3,3}$ :

```
> G := Graph< 6 | [ {3, 4, 5}, {3, 4, 5, 6}, {1, 2, 5, 6},
> {1, 2, 5, 6}, {1, 2, 3, 4, 6}, {2, 3, 4, 5} ] >;
> G;
Graph
Vertex Neighbours
1      3 4 5 ;
2      3 4 5 6 ;
3      1 2 5 6 ;
4      1 2 5 6 ;
5      1 2 3 4 6 ;
6      2 3 4 5 ;
> IsPlanar(G);
false
```

```

> b, 0 := IsPlanar(G);
> IsSubgraph(G, 0);
true
> 0;
Graph
Vertex Neighbours
1      4 5 3 ;
2      3 5 4 ;
3      1 6 2 ;
4      2 6 1 ;
5      6 1 2 ;
6      5 4 3 ;
> IsHomeomorphic(0 : Graph := "K33");
true

```

---

## 149.22 Automorphism Group of a Graph or Digraph

The automorphism group functionality is an interface to B. McKay's *nauty* V 2.2 programme. For a paper describing *nauty* see [McK81]. There also exists a user's manual [McK] describing *nauty*'s essential features.

### 149.22.1 The Automorphism Group Function

**AutomorphismGroup**( $G$  : *parameters*)

Compute the automorphism group of the graph  $G$ . This returns the group  $A$ , the vertex and edge  $G$ -sets (in that order), the power structure  $\mathcal{A}$  of all automorphisms of  $G$ , and the transfer map from  $A$  to  $\mathcal{A}$ . Note that  $G$  may be directed or undirected. For small graphs (i.e. having less than 500 vertices) the canonically labelled graph is also returned.

The automorphism group computation may be driven by several user parameters. There are four principal parameters: **Canonical**, **Stabilizer**, **Invariant** and **Print**. Some of these parameters have associated parameters.

**Canonical**

BOOLELT

*Default* : false

If the parameter **Canonical** is set to **true**, then the canonically labelled graph will be returned. If the graph has fewer than 500 vertices then the default value for this parameter is **true**, while if the graph has 500 or more vertices, the default value is **false**. It is important to note that the canonically labelled graph is *dependent* upon all the parameters used to compute the automorphism group.

**Stabilizer**

[ { GRPHVERT } ]

*Default* :

If the parameter **Stabilizer** is assigned a partition  $P$  of the vertex-set of  $G$  as its value, then the subgroup of the automorphism group of  $G$  that preserves the

partition  $P$  will be computed. The partition given as **Stabilizer** need not be a partition of the full vertex-set of  $G$ . Any vertices not covered by the sets in **Stabilizer** are assumed to belong to an extra partition cell.

<b>Invariant</b>	MONSTGELT	<i>Default :</i>
------------------	-----------	------------------

Use the named invariant to assist the auto group computation. The invariant is specified by a string which is the name of a C function computing an invariant, as on pages 16–17 of the tt nauty manual [McK]. The default value for **Invariant** is "Null" when  $G$  is an undirected graph, or "adjacencies" when  $G$  is a digraph. For a more detailed discussion on invariants see Subsection 149.22.2.

There are three optional associated parameters to **Invariant**:

<b>Minlevel</b>	RNGINTELT	<i>Default : 0</i>
-----------------	-----------	--------------------

<b>Maxlevel</b>	RNGINTELT	<i>Default :</i>
-----------------	-----------	------------------

<b>Arg</b>	RNGINTELT	<i>Default :</i>
------------	-----------	------------------

An expression, depending upon the type of invariant. When  $G$  is undirected these three parameters take the values 0, 1, and 0 respectively. When  $G$  is a digraph they take the values 0, 2, and 0 respectively. When **Invariant** is one of "ind-sets", "cliques", "cellcliq", or "cellind", the default value for **Arg** is 3. Again, see Subsection 149.22.2 for more information.

<b>Print</b>	RNGINTELT	<i>Default : 0</i>
--------------	-----------	--------------------

The integer valued parameter **Print** is used to control informative printing according to the following table:

**Print** := 0 : No output (default).

**Print** := 1 : Statistics are printed.

**Print** := 2 : Summary upon completion of each level.

**Print** := 3 : Print the automorphisms as they are discovered.

<b>IgnoreLabels</b>	BOOLELT	<i>Default : false</i>
---------------------	---------	------------------------

By default, the automorphism group computation respects vertex labels (colours). That is, elements of the group will only take a vertex to an identically labelled vertex. The Boolean value **IgnoreLabels** will treat all vertices as identically labelled for the purposes of this computation.

### 149.22.2 nauty Invariants

Invariants can be used to supplement the built-in partition refinement code in **nauty**, to assist in the processing of difficult graphs. Invariants are used with three parameters:

**Minlevel**: the minimum depth in the search tree at which an invariant is to be applied (default 0)

**Maxlevel**: the maximum depth in the search tree at which an invariant is to be applied (default 1 for undirected graphs and 2 for digraphs)

**Arg**: an integer argument which has a different meaning (or no meaning) for each invariant

The root of the tree, corresponding to the partition provided by the user, has level 1. Note that the invariants use an existing partition and attempt to refine it. Thus, it can be that an invariant fails to refine the partition at the root of the search tree but succeeds further down the tree. In particular, all invariants necessarily fail at the root of the tree if the graph is vertex-transitive.

The invariants are :

"default"

Default — No invariant is used.

"twopaths"

Each vertex is classified according to the vertices reachable from along a path of length 2. This is a cheap invariant sometimes useful for regular graphs. **Arg** is not used.

"adjtriang"

This counts the number of common neighbours between pairs of adjacent (if **Arg** = 0) or non-adjacent (if **Arg** = 1) vertices. This is a fairly cheap invariant which can often work for strongly-regular graphs.

"triples"

This considers in detail the neighbourhoods of each triple of vertices. It often works for strongly-regular graphs that "adjtriang" fails on, but is more expensive. **Arg** is not used.

"quadruples"

Similar to triples but using quadruples of vertices. Much more expensive but can work on graphs with highly regular structure that "triples" fails on. **Arg** is not used.

"celltrips"

Like "triples", but only triples inside one cell are used. One to try for the bipartite incidence graphs of structures like block designs (but won't work for projective planes). **Arg** is not used.

"cellquads"

Like "quadruples", but only quadruples inside one cell are used. Originally designed for some exceptional graphs derived from Hadamard matrices (where applying it at level 2 is best). **Arg** is not used.

"cellquins"

Considers 5-tuples of vertices inside a cell. Very expensive. We don't know of a good application. **Arg** is not used.

"cellfano"

"cellfano2"

These are intended for the bipartite graphs of projective planes and similar highly regular structures. "cellfano2" is cheaper, so try it first. **Arg** is not used. The method is similar to counting Fano subplanes.

"distances"

This uses the distance matrix of the graph. It is good at partitioning general regular graphs (but not strongly regular graphs). Moderately cheap. **Arg** is unused.

"indsets"

This uses the independent sets of size **Arg** (default 3). Can be successful for strongly regular graphs (for  $\text{Arg} \geq 4$ ) or regular bipartite graphs.

"cliques"

This uses the cliques of size **Arg** (default 3). Can be successful for strongly regular graphs (for  $\text{Arg} \geq 4$ ).

"cellind"

This uses the independent sets of size **Arg** (default 3) that lie entirely within one cell of the partition. Try applying at level 2 for difficult vertex-transitive graphs.

"cellcliq"

This uses the cliques of size **Arg** (default 3) that lie entirely within one cell of the partition. Try applying at level 2 for difficult vertex-transitive graphs.

"adjacencies"

This is a simple invariant that corrects for a deficiency in the built-in partition refinement procedure for directed graphs. It is the default in MAGMA for directed graphs. Apply at a few levels, say levels 1 – 5. **Arg** is unused.

These invariants are further described on pages 16–17 of the **nauty** manual [McK]. Also, we provide a facility enabling users to test if a specific **Invariant** achieves a partition refinement for a graph  $G$ .

<b>IsPartitionRefined</b> ( $G$ : <i>parameters</i> )
---

Returns **true** if and only the invariant in **Invariant** could refine the graph  $G$ 's vertex-set partition. If no invariant is set in **Invariant** then the refinement procedure which is tested is taken to be the default one.

Four parameters can be passed:

<b>Stabilizer</b>	[ { VERT } ]	<i>Default :</i>
<b>Invariant</b>	MONSTGELT	<i>Default :</i>
<b>Arg</b>	RNGINTELT	<i>Default :</i>
<b>IgnoreLabels</b>	BOOLELT	<i>Default : false</i>

The parameters have the same meaning as for **AutomorphismGroup**.

### 149.22.3 Graph Colouring and Automorphism Group

For any calls to **nauty** via MAGMA functions the graph colouring of a graph  $G$  (as set by **AssignLabels** or a similar function) is taken as an *intrinsic* property of  $G$ . Consequently, the default automorphism group computed by **nauty** is the group for the coloured graph  $G$ . In particular this implies that, when  $G$  is coloured, this default automorphism group is *not*  $G$ 's full automorphism group.

There are several MAGMA functions which make use of the automorphism group of a graph, and therefore rely on a call to `nauty`. We list them here:

- 1 `Diameter`, `DiameterPath`, `IsDistanceRegular`, `GirthCycle`, `Girth`
- 2 `IsDistanceTransitive`, `IsTransitive`, `IsPrimitive`, `IsSymmetric`, `IsIsomorphic`, `IsEdgeTransitive`, `EdgeGroup`, `OrbitsPartition`, `CanonicalGraph`

The functions in Group 1 make use of the automorphism group of the graph only as a means to improve efficiency. If some group has already been computed for the graph under consideration then this group will be used. Otherwise the graph's default group is computed.

The functions in Group 2 all use the graph's default group. If a group  $A$  of the graph  $G$  has already been computed but is not  $G$ 's default group (for example if a stabilizer of  $G$  was given when computing  $A$ ) then the graph's default group will be computed. Consequently, as an example, if  $G$  is coloured, `IsTransitive( $G$ )` will (always) return `false`.

For all these functions it is possible to drive the group computation by setting an appropriate `Invariant` parameter. This is particularly useful when it is known that some invariant will speed up the group computation. This is achieved by first computing the group using `AutomorphismGroup` while setting `Invariant`. This parameter is then remembered for any further (re)computation of the default automorphism group, unless reset by a subsequent call to `AutomorphismGroup`. For convenience, the function `IsIsomorphic` accepts a parameter suite very similar to that of the `AutomorphismGroup` function: It is particularly useful when stabilizers need to be set.

#### 149.22.4 Variants of Automorphism Group

##### CanonicalGraph( $G$ )

Given a graph  $G$ , return the canonically labelled graph isomorphic to  $G$ . To see which automorphism group is computed see Subsection 149.22.3. Users should be aware that the canonical graph is *dependent* upon the invariant used when computing the automorphism group. Thus, the computation of the automorphism group of  $G$  using different invariants will produce *different* canonical labellings.

##### EdgeGroup( $G$ )

Returns the automorphism group of the graph  $G$  in its action on the edges of  $G$ , and the  $G$ -set of edges of  $G$ . To see which automorphism group is computed see Subsection 149.22.3.

##### IsIsomorphic( $G$ , $H$ : *parameters*)

This function returns `true` if the graphs  $G$  and  $H$  are isomorphic. If  $G$  and  $H$  are isomorphic, a second value is returned, namely a mapping between the vertices of  $G$  and the vertices of  $H$  giving the isomorphism.

The parameters accepted by `IsIsomorphic` are almost the same as for `AutomorphismGroup`. The exceptions are:



**Stabilizer** [ { VERT } ] *Default :*

This sets the stabilizer set for *both* graphs  $G$  and  $H$ . If one wishes to set two distinct stabilizers then one would also use:

**Stabilizer2** [ { VERT } ] *Default :*

This sets the stabilizer set for the second graph  $H$  only.

**IgnoreLabels** BOOLELT *Default : false*

This indicates whether or not to ignore the graph labelling for *both* graphs  $G$  and  $H$ .

Graph isomorphism is understood as mapping colours to colours and stabilizers to stabilizers. Consequently comparing two graphs for isomorphism will always return **false** if one graph is coloured while the other one is not. Similarly two graphs can't be isomorphic if each is coloured using a different set of colours. Moreover, it is a **nauty** requirement that in order to achieve mapping of stabilizers they must be compatible: that is, they *must* have same-sized cells in the same order.

In general, to see which and how the graphs' automorphism groups are computed see Subsection 149.22.3.

---

### Example H149E18

We provide a very simple example of the use of **AutomorphismGroup** which shows how the automorphism group and the corresponding canonical graph differ depending on the labelling of the graph and of the stabilizer given, if any.

```
> G := Graph<5 | { {1,2}, {2,3}, {3,4}, {4,5}, {5,1} }>;
>
> AssignLabels(VertexSet(G), ["a", "b", "a", "b", "b"]);
> A, _, _, _, C1 := AutomorphismGroup(G);
> A;
Permutation group A acting on a set of cardinality 5
Order = 2
(1, 3)(4, 5)
> C1;
Graph
Vertex Neighbours
1      3 5 ;
2      4 5 ;
3      1 4 ;
4      2 3 ;
5      1 2 ;
> B, _, _, _, C2 := AutomorphismGroup(G : IgnoreLabels := true);
> B;
Permutation group B acting on a set of cardinality 5
Order = 10 = 2 * 5
(2, 5)(3, 4)
(1, 2)(3, 5)
> C2;
```

```

Graph
Vertex Neighbours
1      2 3 ;
2      1 4 ;
3      1 5 ;
4      2 5 ;
5      3 4 ;
> C, _, _, _, C3 := AutomorphismGroup(G : Stabilizer :=
> [ { VertexSet(G) | 1, 2 } ]);
> C;
Permutation group C acting on a set of cardinality 5
> #C;
1
> C3;
Graph
Vertex Neighbours
1      3 4 ;
2      3 5 ;
3      1 2 ;
4      1 5 ;
5      2 4 ;

```

We now check that the graph returned by `CanonicalGraph` is identical to  $C_1$ :

```

> C4 := CanonicalGraph(G);
> C4 eq C1;
true

```

---

### Example H149E19

These two examples should help clarify how `IsIsomorphic` behaves for graphs for which a colouring is defined. The first example shows that two graphs cannot be isomorphic if one is coloured while the other is not. This example also demonstrates the use of stabilizers.

```

> G1 := CompleteGraph(5);
> AssignLabels(VertexSet(G1), ["a", "a", "b", "b", "b"]);
> G2 := CompleteGraph(5);
> IsIsomorphic(G1, G2);
false
> IsIsomorphic(G1, G2 : IgnoreLabels := true);
true
>
> V1 := Vertices(G1);
> V2 := Vertices(G2);
> S1 := { V1 | 1, 2, 3 };
> S2 := { V2 | 3, 4, 5 };
>
> IsIsomorphic(G1, G2 : Stabilizer := [S1],
>           Stabilizer2 := [S2]);

```

```

false
> IsIsomorphic(G1, G2 : Stabilizer := [S1],
>      Stabilizer2 := [S2], IgnoreLabels := true);
true
> IsIsomorphic(G1, G2 : Stabilizer := [S1],
> IgnoreLabels := true);
true
>
> SS1 := [ { V1 | 1}, {V1 | 2, 3} ];
> SS2 := [ { V2 | 3, 4}, { V2 | 1} ];
> IsIsomorphic(G1, G2 : Stabilizer := SS1, Stabilizer2 := SS2,
> IgnoreLabels := true);
false
>
> SS1 := [ {V1 | 2, 3}, { V1 | 1} ];
> IsIsomorphic(G1, G2 : Stabilizer := SS1, Stabilizer2 := SS2,
> IgnoreLabels := true);
true

```

The second example shows that two graphs are isomorphic if and only if colours map to the *same* colours.

```

> G1 := CompleteGraph(5);
> AssignLabels(VertexSet(G1), ["b", "b", "b", "a", "a"]);
> G2 := CompleteGraph(5);
> AssignLabels(VertexSet(G2), ["a", "a", "b", "b", "b"]);
> IsIsomorphic(G1, G2);
true
>
> G1 := CompleteGraph(5);
> AssignLabels(VertexSet(G1), ["a", "b", "b", "c", "c"]);
> G2 := CompleteGraph(5);
> AssignLabels(VertexSet(G2), ["a", "c", "b", "b", "c"]);
> IsIsomorphic(G1, G2);
true
>
> G1 := CompleteGraph(5);
> G2 := CompleteGraph(5);
> AssignLabels(VertexSet(G1), ["a", "a", "b", "b", "b"]);
> AssignLabels(VertexSet(G2), ["b", "b", "a", "a", "a"]);
> IsIsomorphic(G1, G2);
false
> IsIsomorphic(G1, G2 : IgnoreLabels := true);
true
>
> G1 := CompleteGraph(5);
> AssignLabels(VertexSet(G1), ["b", "b", "b", "a", "a"]);
> G2 := CompleteGraph(5);
> AssignLabels(VertexSet(G2), ["b", "b", "c", "c", "c"]);

```

```
> IsIsomorphic(G1, G2);
false
> IsIsomorphic(G1, G2 : IgnoreLabels := true);
true
```

---

### 149.22.5 Action of Automorphisms

The automorphism group  $A$  of a graph  $G$  is given in its action on the standard support and it does not act directly on  $G$ . The action of  $A$  on  $G$  is obtained using the  $G$ -set mechanism. The two basic  $G$ -sets associated with the graph correspond to the action of  $A$  on the set of vertices  $V$  and the set of edges  $E$  of  $G$ . These two  $G$ -sets are given as return values of the function **AutomorphismGroup** or may be constructed directly. Additional  $G$ -sets associated with a graph may be built using the  $G$ -set constructors. Given a  $G$ -set  $Y$  for  $A$ , the action of  $A$  on  $Y$  may be studied using the permutation group functions that allow a  $G$ -set as an argument. In this section, only a few of the available functions are described: see the section on  $G$ -sets for a complete list.

**Image(a, Y, y)**

Let  $a$  be an element of the automorphism group  $A$  for the graph  $G$  and let  $Y$  be a  $G$ -set for  $A$ . Given an element  $y$  belonging either to  $Y$  or to a  $G$ -set derived from  $Y$ , find the image of  $y$  under  $a$ .

**Orbit(A, Y, y)**

Let  $A$  be a subgroup of the automorphism group for the graph  $G$  and let  $Y$  be a  $G$ -set for  $A$ . Given an element  $y$  belonging either to  $Y$  or to a  $G$ -set derived from  $Y$ , construct the orbit of  $y$  under  $A$ .

**Orbits(A, Y)**

Let  $A$  be a subgroup of the automorphism group for the graph  $G$  and let  $Y$  be a  $G$ -set for  $G$ . This function constructs the orbits of the action of  $A$  on  $Y$ .

**Stabilizer(A, Y, y)**

Let  $A$  be a subgroup of the automorphism group for the graph  $G$  and let  $Y$  be a  $G$ -set for  $A$ . Given an element  $y$  belonging either to  $Y$  or to a  $G$ -set derived from  $Y$ , construct the stabilizer of  $y$  in  $A$ .

**Action(A, Y)**

Given a subgroup  $A$  of the automorphism group of the graph  $G$ , and a  $G$ -set  $Y$  for  $A$ , construct the homomorphism  $\phi : A \rightarrow L$ , where the permutation group  $L$  gives the action of  $A$  on the set  $Y$ . The function returns:

- (a) The natural homomorphism  $\phi : A \rightarrow L$ ;
- (b) The induced group  $L$ ;
- (c) The kernel of the action (a subgroup of  $A$ ).

**ActionImage(A, Y)**

Given a subgroup  $A$  of the automorphism group of the graph structure  $G$ , and a  $G$ -set  $Y$  for  $A$ , construct the permutation group  $L$  giving the action of  $A$  on the set  $Y$ .

**ActionKernel(A, Y)**

Given a subgroup  $A$  of the automorphism group of the graph  $G$ , and a  $G$ -set  $Y$  for  $A$ , construct the kernel of the action of  $A$  on the set  $Y$ .

**Example H149E20**

We construct the Clebsch graph (a strongly regular graph) and investigate the action of its automorphism group.

```
> S := { 1 .. 5 };
> V := &join[ Subsets({1..5}, 2*k) : k in [0..#S div 2]];
> E := { {u,v} : u,v in V | #(u sdiff v) eq 4 };
> G, V, E := StandardGraph( Graph< V | E >);
> G;
Graph
Vertex  Neighbours
1       4 6 8 10 12 ;
2       5 6 9 12 14 ;
3       9 10 12 13 16 ;
4       1 7 9 14 16 ;
5       2 7 8 10 16 ;
6       1 2 13 15 16 ;
7       4 5 12 13 15 ;
8       1 5 9 11 13 ;
9       2 3 4 8 15 ;
10      1 3 5 14 15 ;
11      8 12 14 15 16 ;
12      1 2 3 7 11 ;
13      3 6 7 8 14 ;
14      2 4 10 11 13 ;
15      6 7 9 10 11 ;
16      3 4 5 6 11 ;
> A, AV, AE := AutomorphismGroup(G);
> A;
Permutation group aut acting on a set of cardinality 16
Order = 1920 = 2^7 * 3 * 5
(2, 15)(5, 11)(7, 14)(10, 12)
(3, 11)(8, 10)(9, 14)(13, 15)
(2, 11)(5, 15)(6, 8)(9, 16)
(2, 7)(4, 6)(9, 13)(14, 15)
(1, 2, 13, 15)(3, 11, 4, 5)(7, 10, 12, 14)(8, 9)
> CompositionFactors(A);
G
```

```

      | Cyclic(2)
      *
      | Alternating(5)
      *
      | Cyclic(2)
      *
      | Cyclic(2)
      *
      | Cyclic(2)
      *
      | Cyclic(2)
      1
> IsPrimitive(A);
true

```

From the composition factors we guess that the group is  $Sym(5)$  extended by the elementary abelian group of order 16. Let us verify this and relate it to the graph. We begin by trying to get at the group of order 16. We ask for the Fitting subgroup.

```

> F := FittingSubgroup(A);
> F;
Permutation group F acting on a set of cardinality 16
Order = 16 = 2^4
(1, 13)(2, 11)(3, 4)(5, 15)(6, 8)(7, 10)(9, 16)(12, 14)
(1, 10)(2, 9)(3, 12)(4, 14)(5, 8)(6, 15)(7, 13)(11, 16)
(1, 11)(2, 13)(3, 5)(4, 15)(6, 14)(7, 9)(8, 12)(10, 16)
(1, 12)(2, 6)(3, 10)(4, 7)(5, 16)(8, 11)(9, 15)(13, 14)

```

Since  $A$  is primitive, this looks as if it is an elementary abelian regular normal subgroup:

```

> EARNs(A) eq F;
true

```

Thus,  $A$  has a regular normal subgroup of order 16. Further, it acts transitively on the vertices. The complement of  $N$  is the stabilizer of a point.

```

> S1 := Stabilizer(A, AV, V!1);
> #S1;
120
> IsTransitive(S1);
false
> O := Orbits(S1);
> O;
[
  GSet{ 1 },
  GSet{ 2, 3, 5, 7, 9, 11, 13, 14, 15, 16 },
  GSet{ 4, 6, 8, 10, 12 }
]
> IsSymmetric(ActionImage(S1, O[3]));

```

`true`

So the stabilizer of the vertex 1 is  $\text{Sym}(5)$ . Note that it acts faithfully on the 5 neighbours of 1.

---

## 149.23 Symmetry and Regularity Properties of Graphs

`IsTransitive(G)`

`IsVertexTransitive(G)`

Returns `true` if the automorphism group of the graph  $G$  is transitive, otherwise `false`. To see which automorphism group is computed see Subsection 149.22.3.

`IsEdgeTransitive(G)`

Returns `true` if the automorphism group of the graph  $G$  is transitive on the edges of  $G$  (i.e. if the edge group of  $G$  is transitive). To see which automorphism group is computed see Subsection 149.22.3.

`OrbitsPartition(G)`

Given a graph  $G$ , return the partition of its vertex-set corresponding to the orbits of its automorphism group in the form of a set system. To see which automorphism group is computed see Subsection 149.22.3.

`IsPrimitive(G)`

Returns `true` if the graph  $G$  is primitive, i.e. if its automorphism group is primitive. To see which automorphism group is computed see Subsection 149.22.3.

`IsSymmetric(G)`

Returns `true` if the graph  $G$  is symmetric, i.e. if for all pairs of vertices  $u, v$  and  $w, t$  such that  $u \text{ adj } v$  and  $w \text{ adj } t$ , there exists an automorphism  $a$  such  $u^a = w$  and  $v^a = t$ . To see which automorphism group is computed see Subsection 149.22.3.

`IsDistanceTransitive(G)`

Returns `true` if the connected graph  $G$  is distance transitive i.e. if for all vertices  $u, v, w, t$  of  $G$  such that  $d(u, v) = d(w, t)$ , there is an automorphism  $a$  in  $A$  such that  $u^a = w$  and  $v^a = t$ . To see which automorphism group is computed see Subsection 149.22.3.

`IsDistanceRegular(G)`

Returns `true` if the graph  $G$  is distance regular, otherwise `false`. To see how the automorphism group of  $G$  is computed see Subsection 149.22.3.

**IntersectionArray(G)**

The intersection array of the distance regular graph  $G$ . This is returned as a sequence  $[k, b(1), \dots, b(d-1), 1, c(2), \dots, c(d)]$  where  $k$  is the valency of the graph,  $d$  is the diameter of the graph, and the numbers  $b(i)$  and  $c(i)$  are defined as follows: Let  $N_j(u)$  denote the set of vertices of  $G$  that lie at distance  $j$  from vertex  $u$ . Let  $u$  and  $v$  be a pair of vertices satisfying  $d(u, v) = j$ .

Then  $c(j) = \text{number of vertices in } N_{j-1}(v) \text{ that are adjacent to } u, (1 \leq j \leq d)$ , and  $b(j) = \text{number of vertices in } N_{j+1}(v) \text{ that are adjacent to } u (0 \leq j \leq d-1)$ .

**Example H149E21**

We illustrate the use of some of the symmetry functions by applying them to the graph of the 8-dimensional cube.

```
> g := KCubeGraph(8);
> IsVertexTransitive(g);
true
> IsEdgeTransitive(g);
true
> IsSymmetric(g);
true
> IsDistanceTransitive(g);
true
> IntersectionArray(g);
[ 8, 7, 6, 5, 4, 3, 2, 1, 1, 2, 3, 4, 5, 6, 7, 8 ]
```

We also see that the functions using the graph's automorphism group are dependent upon the graph being coloured or not:

```
> q := 9;
> P := FiniteProjectivePlane(q);
> X := IncidenceGraph(P);
>
> Order(X);
182
> Valence(X);
10
> Diameter(X);
3
> Girth(X);
6
> O1 := OrbitsPartition(X);
> IsSymmetric(X);
true
>
> Labels := [ "a" : i in [1..96] ];
> #Labels;
96
```



```

> AssignLabels(VertexSet(X), Labels);
> O2 := OrbitsPartition(X);
> O2 eq O1;
false
> IsSymmetric(X);
false

```

---

## 149.24 Graph Databases and Graph Generation

MAGMA provides interfaces to some databases of certain graphs of interest. These databases are not provided by default with MAGMA, but may be downloaded from the optional databases section of the MAGMA website.

### 149.24.1 Strongly Regular Graphs

A catalogue of strongly regular graphs is available. This catalogue has been put together from various sources by B. McKay and can be found at

<http://cs.anu.edu.au/~bdm/data/>

Graphs in the database are indexed by a sequence of four parameters. They are, in order: the order of the graph, its degree, the number of common neighbours to each pair of adjacent vertices, and the number of common neighbours to each pair of non-adjacent vertices.

**StronglyRegularGraphsDatabase()**

Opens the database of strongly regular graphs.

**Classes(D)**

Returns all the parameter sequences used to index the graphs in the database  $D$ .

**NumberOfClasses(D)**

Returns the number of “classes” of graphs in the database  $D$ .

**NumberOfGraphs(D)**

Returns the number of graphs in the database  $D$ .

**NumberOfGraphs(D, S)**

Returns the number of graphs in the database  $D$  with parameter sequence  $S$ .

**Graphs(D, S)**

Returns (in a sequence) all the graphs in the database  $D$  with parameter sequence  $S$ .

Graph( <i>D</i> , <i>S</i> , <i>i</i> )
---

Returns the *i*th graph in the database *D* with parameter sequence *S*.

RandomGraph( <i>D</i> )
-------------------------

Returns a random graph in the database *D*.

RandomGraph( <i>D</i> , <i>S</i> )
------------------------------------

Returns a random graph in the database *D* with parameter sequence *S*.

for <i>G</i> in <i>D</i> do ... end for;
--

The database of strongly regular graphs may appear as the range in the **for**-statement.

---

### Example H149E22

The following few statements illustrate the basic access functions to the database of strongly regular graphs.

```
> D := StronglyRegularGraphsDatabase();
> Cs := Classes(D);
> Cs;
[
  [ 25, 8, 3, 2 ],
  [ 25, 12, 5, 6 ],
  [ 26, 10, 3, 4 ],
  [ 27, 10, 1, 5 ],
  [ 28, 12, 6, 4 ],
  [ 29, 14, 6, 7 ],
  [ 35, 16, 6, 8 ],
  [ 35, 14, 4, 6 ],
  [ 36, 15, 6, 6 ],
  [ 37, 18, 8, 9 ],
  [ 40, 12, 2, 4 ]
]
> assert NumberOfClasses(D) eq #Cs;
>
> NumberOfGraphs(D);
43442
>
> for i in [1..#Cs] do
>   NumberOfGraphs(D, Cs[i]);
> end for;
1
15
10
1
4
41
```

```

3854
180
32548
6760
28
>
> gs := Graphs(D, Cs[2]);
>
> g := Graph(D, Cs[2], Random(1, NumberOfGraphs(D, Cs[2])));

```

---

### 149.24.2 Small Graphs

An enumeration of small graphs with various properties has been created by B. McKay and may be found at

<http://cs.anu.edu.au/~bdm/data/graphs.html>

Certain of these databases are available from within MAGMA: Simple graphs, Eulerian graphs, planar connected graphs, and self-complementary graphs.

#### 149.24.2.1 Creation of Small Graph Databases

SmallGraphDatabase( $n$  : *parameters*)

**IncludeDisconnected**      **BOOL**

*Default* : **false**

Opens the database of simple graphs with  $n$  vertices,  $2 \leq n \leq 10$ . If the optional parameter **IncludeDisconnected** is set to true then the database will also include non-connected graphs.

EulerianGraphDatabase( $n$  : *parameters*)

**IncludeDisconnected**      **BOOL**

*Default* : **false**

Opens the database of Eulerian graphs with  $n$  vertices,  $3 \leq n \leq 11$ . If the optional parameter **IncludeDisconnected** is set to true then the database will also include non-connected graphs. The allowed range of  $n$  for non-connected graphs is  $2 \leq n \leq 12$ .

PlanarGraphDatabase( $n$ )

Opens the database of planar connected graphs with  $n$  vertices,  $2 \leq n \leq 11$ .

SelfComplementaryGraphDatabase( $n$ )

Opens the database of self-complementary graphs with  $n$  vertices,  $n \in \{4, 5, 8, 9, 12, 13, 16, 17, 20\}$ . For  $n = 20$  this is not a complete enumeration.

### 149.24.2.2 Access functions

#D

Returns the number of graphs in the database  $D$ .

Graph( $D$ ,  $i$ )

Returns the  $i$ th graph in the database  $D$ .

Random( $D$ )

Returns a random graph from the database  $D$ .

for  $G$  in  $D$  do ... end for;

A database of small graphs may appear as the range in the **for**-statement.

### 149.24.3 Generating Graphs

We provide an interface to a graph generation programme, also due to B. McKay (see [McK98]). For the time being, users wanting to benefit from this facility must download the generation programme themselves directly from

<http://cs.anu.edu.au/~bdm/nauty/>

**Important restriction:** Since this program runs within a Unix pipe it is only available to users running MAGMA on a Unix platform.

**And a note of caution:** When the graph generation programme is used to generate reasonably large graphs ( $n > 17$ ) it can be observed that the procedure of closing down the pipe (ie. closing the stream) may take some time. This will happen when the closing down attempt is made before the programme has completed the generation of all the graphs.

GenerateGraphs( $n$  : *parameters*)

Opens a pipe to the graph generation programme to generate all graphs of order  $n$ . Only available on Unix platforms.

The **GenerateGraphs** function allows the user to drive the generation programme via a set of parameters. These parameters are described below.

Once the generation programme has been downloaded from

<http://cs.anu.edu.au/~bdm/nauty/>

and compiled (using **make geng**), the resulting executable (named **geng** – it is *compulsory* that the resulting executable's name be **geng**) can be placed anywhere in the user's directory tree. The environment variable **MAGMA\_NAUTY** *must* then be set to the path where the executable/command **geng** is to be found.

<b>FirstGraph</b>	RNGINTELT	<i>Default : 1</i>
-------------------	-----------	--------------------

Reading of the generated graphs starts at the **FirstGraph**-th graph.

<b>MinEdges</b>	RNGINTELT	<i>Default :</i>
-----------------	-----------	------------------

Generate graphs with minimum number of edges **MinEdges**.

<b>MaxEdges</b>	RNGINTELT	<i>Default :</i>
Generate graphs with maximum number of edges <b>MaxEdges</b> .		
<b>Classes</b>	RNGINTELT	<i>Default : 1</i>
Divide the generated graphs into disjoint <b>Classes</b> classes of very approximately equal size.		
<b>Class</b>	RNGINTELT	<i>Default : 1</i>
When generated graphs are divided into disjoint <b>Classes</b> classes, write only the <b>Class</b> th class.		
<b>Connected</b>	BOOLELT	<i>Default : false</i>
Only generate connected graphs.		
<b>Biconnected</b>	BOOLELT	<i>Default : false</i>
Only generate biconnected graphs.		
<b>TriangleFree</b>	BOOLELT	<i>Default : false</i>
Only generate triangle-free graphs.		
<b>FourCycleFree</b>	BOOLELT	<i>Default : false</i>
Only generate 4-cycle-free graphs.		
<b>Bipartite</b>	BOOLELT	<i>Default : false</i>
Only generate bipartite graphs.		
<b>MinDeg</b>	RNGINTELT	<i>Default :</i>
Specify a lower bound for the minimum degree.		
<b>MaxDeg</b>	RNGINTELT	<i>Default :</i>
Specify an upper bound for the maximum degree.		
<b>Canonical</b>	BOOLELT	<i>Default : false</i>
Canonically label output graphs.		
<b>SparseRep</b>	BOOLELT	<i>Default : false</i>
If <b>true</b> , generate the graphs in Sparse6 format (see below).		

**NextGraph**(*F*: *parameters*)

<b>SparseRep</b>	BOOL	<i>Default : false</i>
------------------	------	------------------------

Returns **true** if and only if file *F* is not at the end. In this case the next graph is returned as well.

Since **NextGraph** can in principle take as an argument any MAGMA object of type **File**, the following restriction applies to the **File** *F*: The graphs in *F* *must* be in either of the output formats Graph6 or Sparse6. Details on the Graph6 and Sparse6 format can be found at

<http://cs.anu.edu.au/~bdm/data/formats.html>

If **SparseRep** is **true** then the resulting graph will have a sparse representation. This of course is of special interest if the graphs read from *F* are also in Sparse6 format.

**Example H149E23**

---

The following statements should help clarify the usage of the graph generation programme.

```
> F := GenerateGraphs (12:
>   FirstGraph:= 10,
>   Connected:= true,
>   Biconnected:= true,
>   TriangleFree:= true,
>   FourCycleFree:= true,
>   Bipartite:= true,
>   MinDeg:= 1,
>   MaxDeg:= 9
> );
```

We'll read all the graphs from the 10th graph onwards (one can check that 28 graphs have been generated):

```
> count := 0;
> while true do
>   more := NextGraph(F);
>   if more then
>     count += 1;
>   else
>     break;
>   end if;
> end while;
> count;
19
```

If one wants to work with sparse graphs, it is recommended to proceed as follows:

```
> F := GenerateGraphs (6: SparseRep := true);
> count := 0;
> while true do
>   more := NextGraph(F: SparseRep := true);
>   if more then
>     count += 1;
>   else
>     break;
>   end if;
> end while;
> count;
156
```

---

### 149.24.4 A General Facility

In order to give users more flexibility in dealing with certain graph files the MAGMA function `OpenGraphFile` is provided. It allows one to open either a graph file or a pipe to a graph generation programme. Since in both cases (file or Unix pipe) the outcome is the access to a stream of graphs, we henceforth refer to the graphs to be read as a *graph stream*.

**The usual restriction:** The `OpenGraphFile` which opens a pipe is only available to users running MAGMA on a Unix platform.

**Accessing and reading the graph stream:** Reading the graph stream is achieved by the above described access function `NextGraph`. As mentioned there, the graphs in the graph stream *must* be in either of the output formats Graph6 or Sparse6. This is why `OpenGraphFile` is restricted to streams of graphs in format Graph6 or Sparse6.

Details on the Graph6 and Sparse6 format can be found at

<http://cs.anu.edu.au/~bdm/data/formats.html>.

<code>OpenGraphFile(s, f, p)</code>
-------------------------------------

Opens a graph file/pipe at position  $p$ . If the stream to be opened is a Unix pipe then the string  $s$  *must* have the format “cmd command” where command stands for the command to run including necessary parameters. If the stream to be opened is a file the string  $s$  has format “filename”.

The integer  $f$  indicates that the record length is fixed, which is true for streams in Graph6 format with every graph having the same order. This permits rapid positioning to position  $p$  in that case. If in doubt, use  $f = 0$ . Also, positioning to 0 or positioning to 1 has the same effect of positioning to the start of the stream.

Opening a pipe is only available on Unix platforms. The file/pipe  $F$  *must* contain graphs in Graph6 and Sparse6 format.

---

#### Example H149E24

As an example one could download one of the files found at

<http://cs.anu.edu.au/~bdm/data/>

Assuming this has been done, one can then proceed to read the graphs in the file:

```
> F := OpenGraphFile("/home/paule/graph/bdm_data/sr251256.g6", 0, 0);
>
> count := 0;
> more, g := NextGraph(F);
> while more do
>   count += 1;
>   more, g := NextGraph(F);
> end while;
> count;
15
```

Alternatively one could also drive the graph generation programme (or any other suitable programme for that matter) described in 149.24.3 using the `OpenGraphFile` access function.

The graph generation programme's name is `geng` (which can be found at <http://cs.anu.edu.au/~bdm/nauty/>) and has a help facility:

```
> F := OpenGraphFile("cmd /home/paule/graph/bdm_pgr/nauty/geng -help", 0, 0);
Usage: geng [-cCmtfbd#D#] [-uygsnh] [-lvq] [-x#X#] n [mine[:maxe]] [res/mod]
[file]
Generate all graphs of a specified class.
    n      : the number of vertices (1..32)
mine:maxe : a range for the number of edges
            #:0 means '# or more' except in the case 0:0
res/mod   : only generate subset res out of subsets 0..mod-1
    -c     : only write connected graphs
    -C     : only write biconnected graphs
    -t     : only generate triangle-free graphs
    -f     : only generate 4-cycle-free graphs
    -b     : only generate bipartite graphs
            (-t, -f and -b can be used in any combination)
    -m     : save memory at the expense of time (only makes a
            difference in the absence of -b, -t, -f and n <= 30).
    -d#    : a lower bound for the minimum degree
    -D#    : a upper bound for the maximum degree
    -v     : display counts by number of edges
    -l     : canonically label output graphs
    -u     : do not output any graphs, just generate and count them
    -g     : use graph6 output (default)
    -s     : use sparse6 output
    -y     : use the obsolete y-format instead of graph6 format
    -h     : for graph6 or sparse6 format, write a header too
    -q     : suppress auxiliary output (except from -v)
See program text for much more information.
```

Finally, here is a typical run of this graph generation programme:

```
> F := OpenGraphFile(
> "cmd /home/paule/graph/bdm_pgr/nauty/geng 15 -cCtfb -v", 0, 0);
>A geng -Ctfb2D14 n=15 e=15-22
>C 4 graphs with 16 edges
>C 45 graphs with 17 edges
>C 235 graphs with 18 edges
>C 294 graphs with 19 edges
>C 120 graphs with 20 edges
>C 13 graphs with 21 edges
>C 1 graphs with 22 edges
>Z 712 graphs generated in 1.38 sec
```

---



## 149.25 Bibliography

- [**BK73**] C. Bron and J. Kerbosch. Finding All Cliques of an Undirected Graph. *Communications of the ACM* 9, 16(9):575–577, 1973.
- [**BM01**] J. Boyer and W. Myrvold. Simplified  $O(n)$  Planarity Algorithms. submitted, 2001.
- [**Bre79**] D. Brelaz. New Methods to Color the Vertices of a Graph. *Communications of the ACM*, 22(9):251–256, 1979.
- [**Chr75**] N. Christofides. *Graph Theory, An Algorithm Approach*. Academic Press, 1975.
- [**Eve79**] Shimon Even. *Graph Algorithms*. Computer Science Press, 1979.
- [**GM01**] C. Gutwenger and P. Mutzel. A Linear Time Implementation of SPQR-Trees. In J. Marks, editor, *Graph Drawing 2000*, volume 1984 of *LNCS*, pages 70–90. Springer-Verlag, 2001.
- [**HT73**] J.E. Hopcroft and R.E. Tarjan. Dividing a Graph into Triconnected Components. *SIAM J. Comput.*, 2(3):135–158, 1973.
- [**Lor89**] P. Lorimer. The construction of Tutte’s 8-cage and the Conder graph. *J. of Graph Theory*, 13(5):553–557, 1989.
- [**McK**] B. D. McKay. **nauty** User’s Guide (Version 2.2).  
URL:<http://cs.anu.edu.au/~bdm/nauty/nug.pdf>.
- [**McK81**] B. D. McKay. Practical Graph Isomorphism. *Congressus Numerantium*, 30: 45–87, 1981.
- [**McK98**] B. D. McKay. Isomorph-free exhaustive generation. *J. Algorithms*, 26:306–324, 1998.
- [**RAO93**] T.L. Magnanti R.K. Ahuja and J.B. Orlin. *Network Flows, Theory, Algorithms, and Applications*. Prentice Hall, 1993.
- [**TCR90**] C.E. Leiserson T.H. Cormen and R.L. Rivest. *Introduction to Algorithms*. MIT Press, 1990.
- [**WM**] N. Walker W. Myrvold, T. Prsa. A Dynamic Programming Approach for Timing and Designing Clique Algorithms. available at  
URL:<http://www.csr.uvic.ca/~wendym/>.



# 150 MULTIGRAPHS

<b>150.1 Introduction . . . . .</b>	<b>5003</b>		
<b>150.2 Construction of Multigraphs</b>	<b>5004</b>		
150.2.1 Construction of a General Multi-graph . . . . .	5004	AssignCapacity( $\sim G$ , $e$ , $c$ )	5012
MultiGraph< >	5004	AssignWeight( $\sim G$ , $e$ , $w$ )	5012
MultiGraph< >	5004	AssignLabels( $\sim G$ , $S$ , $D$ )	5012
150.2.2 Construction of a General Multidigraph . . . . .	5005	AssignCapacities( $\sim G$ , $S$ , $D$ )	5012
MultiDigraph< >	5005	AssignWeights( $\sim G$ , $S$ , $D$ )	5012
MultiDigraph< >	5005	AssignEdgeLabels( $\sim G$ , $D$ )	5013
150.2.3 Printing of a Multi(di)graph . . .	5006	AssignCapacities( $\sim G$ , $D$ )	5013
150.2.4 Operations on the Support . . .	5007	AssignWeights( $\sim G$ , $D$ )	5013
Support( $G$ )	5007	IsLabelled( $e$ )	5013
Support( $V$ )	5007	IsLabelled( $E$ )	5013
ChangeSupport( $G$ , $S$ )	5007	IsEdgeLabelled( $G$ )	5013
ChangeSupport( $\sim G$ , $S$ )	5007	IsCapacitated( $E$ )	5013
StandardGraph( $G$ )	5007	IsEdgeCapacitated( $G$ )	5013
<b>150.3 The Vertex-Set and Edge-Set of Multigraphs . . . . .</b>	<b>5008</b>	IsWeighted( $E$ )	5014
EdgeIndices( $u$ , $v$ )	5008	IsEdgeWeighted( $G$ )	5014
Indices( $u$ , $v$ )	5008	Label( $e$ )	5014
EdgeMultiplicity( $u$ , $v$ )	5008	Capacity( $e$ )	5014
Multiplicity( $u$ , $v$ )	5008	Weight( $e$ )	5014
Edges( $u$ , $v$ )	5008	Labels( $S$ )	5014
IncidentEdges( $u$ )	5008	Capacities( $S$ )	5014
!	5008	Weights( $S$ )	5014
!	5009	Labels( $E$ )	5014
$E . i$	5009	Capacities( $E$ )	5014
EndVertices( $e$ )	5009	Weights( $E$ )	5014
InitialVertex( $e$ )	5009	EdgeLabels( $G$ )	5015
TerminalVertex( $e$ )	5009	EdgeCapacities( $G$ )	5015
Index( $e$ )	5009	EdgeWeights( $G$ )	5015
eq	5009	DeleteLabel( $\sim G$ , $e$ )	5015
		DeleteCapacity( $\sim G$ , $e$ )	5015
		DeleteWeight( $\sim G$ , $e$ )	5015
		DeleteLabels( $\sim G$ , $S$ )	5015
		DeleteCapacities( $\sim G$ , $S$ )	5015
		DeleteWeights( $\sim G$ , $S$ )	5015
		DeleteEdgeLabels( $\sim G$ )	5015
		DeleteCapacities( $\sim G$ )	5015
		DeleteWeights( $\sim G$ )	5015
<b>150.4 Vertex and Edge Decorations</b>	<b>5011</b>	150.4.3 Unlabelled, or Uncapacitated, or Unweighted Graphs . . . . .	5015
150.4.1 Vertex Decorations: Labels . . .	5011	UnlabelledGraph( $G$ )	5015
AssignLabel( $\sim G$ , $u$ , $l$ )	5011	UncapacitatedGraph( $G$ )	5016
AssignLabels( $\sim G$ , $S$ , $L$ )	5011	UnweightedGraph( $G$ )	5016
AssignVertexLabels( $\sim G$ , $L$ )	5011	<b>150.5 Standard Construction for Multigraphs . . . . .</b>	<b>5018</b>
IsLabelled( $u$ )	5011	150.5.1 Subgraphs . . . . .	5018
IsLabelled( $V$ )	5011	sub< >	5018
IsVertexLabelled( $G$ )	5011	150.5.2 Incremental Construction of Multigraphs . . . . .	5020
Label( $u$ )	5011	+	5020
Labels( $S$ )	5011	+=	5021
Labels( $V$ )	5011	AddVertex( $\sim G$ )	5021
VertexLabels( $G$ )	5011	AddVertices( $\sim G$ , $n$ )	5021
DeleteLabel( $\sim G$ , $u$ )	5011	AddVertex( $\sim G$ , $l$ )	5021
DeleteVertexLabels( $\sim G$ )	5012	AddVertices( $\sim G$ , $n$ , $L$ )	5021
150.4.2 Edge Decorations . . . . .	5012		
AssignLabel( $\sim G$ , $e$ , $l$ )	5012		

-	5021	<b>150.6 Conversion Functions . . . 5026</b>
-	5021	<i>150.6.1 Orientated Graphs . . . . . 5027</i>
-=	5021	<b>OrientatedGraph(G) 5027</b>
-=	5021	<i>150.6.2 Converse . . . . . 5027</i>
RemoveVertex( $\sim G$ , v)	5021	<b>Converse(G) 5027</b>
RemoveVertices( $\sim G$ , U)	5021	<i>150.6.3 Converting between Simple Graphs</i>
+	5021	<i>and Multigraphs . . . . . 5027</i>
+	5021	<b>UnderlyingGraph(G) 5027</b>
+	5022	<b>UnderlyingDigraph(G) 5027</b>
+	5022	<b>UnderlyingMultiGraph(G) 5027</b>
+	5022	<b>UnderlyingMultiDigraph(G) 5028</b>
+=	5022	<b>UnderlyingNetwork(G) 5028</b>
+=	5022	<b>150.7 Elementary Invariants and</b>
+=	5022	<b>Predicates for Multigraphs . 5028</b>
+=	5022	<b>Order(G) 5029</b>
+=	5022	<b>NumberOfVertices(G) 5029</b>
+=	5022	<b>Size(G) 5029</b>
AddEdge(G, u, v)	5022	<b>NumberOfEdges(G) 5029</b>
AddEdge(G, u, v, l)	5022	<b>adj 5029</b>
AddEdge(G, u, v, c)	5022	<b>adj 5029</b>
AddEdge(G, u, v, c, l)	5023	<b>notadj 5029</b>
AddEdge( $\sim G$ , u, v)	5023	<b>notadj 5029</b>
AddEdge( $\sim G$ , u, v, l)	5023	<b>in 5029</b>
AddEdge( $\sim G$ , u, v, c)	5023	<b>notin 5029</b>
AddEdge( $\sim G$ , u, v, c, l)	5023	<b>eq 5029</b>
AddEdges(G, S)	5023	<b>IsSubgraph(G, H) 5029</b>
AddEdges(G, S, L)	5023	<b>IsBipartite(G) 5029</b>
AddEdges( $\sim G$ , S)	5023	<b>Bipartition(G) 5030</b>
AddEdges( $\sim G$ , S, L)	5023	<b>IsRegular(G) 5030</b>
-	5023	<b>IsComplete(G) 5030</b>
-	5023	<b>IsEmpty(G) 5030</b>
-	5024	<b>IsNull(G) 5030</b>
-	5024	<b>IsSimple(G) 5030</b>
-=	5024	<b>IsUndirected(G) 5030</b>
-=	5024	<b>IsDirected(G) 5030</b>
-=	5024	<b>150.8 Adjacency and Degree . . . 5030</b>
-=	5024	<i>150.8.1 Adjacency and Degree Functions</i>
RemoveEdge( $\sim G$ , e)	5024	<i>for Multigraphs . . . . . 5031</i>
RemoveEdges( $\sim G$ , S)	5024	<b>Degree(u) 5031</b>
RemoveEdge( $\sim G$ , u, v)	5024	<b>Alldeg(G, n) 5031</b>
<i>150.5.3 Vertex Insertion, Contraction . . 5024</i>		<b>MaximumDegree(G) 5031</b>
InsertVertex(e)	5025	<b>Maxdeg(G) 5031</b>
InsertVertex(T)	5025	<b>MinimumDegree(G) 5031</b>
Contract(e)	5025	<b>Mindeg(G) 5031</b>
Contract(u, v)	5025	<b>DegreeSequence(G) 5031</b>
Contract(S)	5025	<b>Neighbours(u) 5031</b>
<i>150.5.4 Unions of Multigraphs . . . . . 5025</i>		<b>Neighbors(u) 5031</b>
Union(G, H)	5025	<b>IncidentEdges(u) 5031</b>
join	5025	<i>150.8.2 Adjacency and Degree Functions</i>
Union(N, H)	5026	<i>for Multidigraphs . . . . . 5032</i>
join	5026	<b>InDegree(u) 5032</b>
join	5026	<b>OutDegree(u) 5032</b>
EdgeUnion(G, H)	5026	<b>MaximumInDegree(G) 5032</b>
EdgeUnion(N, H)	5026	<b>Maxindeg(G) 5032</b>

MinimumInDegree(G)	5032	VertexSeparator(G : -)	5035
Minindeg(G)	5032	VertexConnectivity(G : -)	5036
MaximumOutDegree(G)	5032	IsKVertexConnected(G, k : -)	5036
Maxoutdeg(G)	5032	EdgeSeparator(G : -)	5036
MinimumOutDegree(G)	5032	EdgeConnectivity(G : -)	5036
Minoutdeg(G)	5032	IsKEdgeConnected(G, k : -)	5036
Degree(u)	5032		
MaximumDegree(G)	5032	<b>150.10 Spanning Trees . . . . . 5037</b>	
Maxdeg(G)	5032	SpanningTree(G)	5037
MinimumDegree(G)	5033	SpanningForest(G)	5037
Mindeg(G)	5033	BreadthFirstSearchTree(u)	5037
Alldeg(G, n)	5033	BFSTree(u)	5037
DegreeSequence(G)	5033	DepthFirstSearchTree(u)	5038
InNeighbours(u)	5033	DFSTree(u)	5038
InNeighbors(u)	5033		
OutNeighbours(u)	5033	<b>150.11 Planar Graphs . . . . . 5038</b>	
OutNeighbors(u)	5033	IsPlanar(G)	5038
IncidentEdges(u)	5033	Obstruction(G)	5038
		IsHomeomorphic(G: -)	5038
<b>150.9 Connectedness . . . . . 5033</b>		Faces(G)	5038
<i>150.9.1 Connectedness in a Multigraph . 5034</i>		Face(u, v)	5039
IsConnected(G)	5034	Face(e)	5039
Components(G)	5034	NFaces(G)	5039
Component(u)	5034	NumberOfFaces(G)	5039
IsSeparable(G)	5034	Embedding(G)	5039
IsBiconnected(G)	5034	Embedding(v)	5039
CutVertices(G)	5034		
Bicomponents(G)	5034	<b>150.12 Distances, Shortest Paths and Minimum Weight Trees . . 5042</b>	
<i>150.9.2 Connectedness in a Multidigraph 5034</i>		Reachable(u, v : -)	5042
IsStronglyConnected(G)	5034	Distance(u, v : -)	5042
IsWeaklyConnected(G)	5034	Distances(u : -)	5042
StronglyConnectedComponents(G)	5034	PathExists(u, v : -)	5043
Component(u)	5034	Path(u, v : -)	5043
<i>150.9.3 Triconnectivity for Multigraphs . 5035</i>		ShortestPath(u, v : -)	5043
IsTriconnected(G)	5035	Paths(u : -)	5043
Splitcomponents(G)	5035	ShortestPaths(u : -)	5043
SeparationVertices(G)	5035	GeodesicExists(u, v : -)	5043
<i>150.9.4 Maximum Matching in Bipartite Multigraphs . . . . . 5035</i>		Geodesic(u, v : -)	5043
MaximumMatching(G : -)	5035	Geodesics(u : -)	5043
<i>150.9.5 General Vertex and Edge Connec- tivity in Multigraphs and Multidi- graphs . . . . . 5035</i>		HasNegativeWeightCycle(u : -)	5043
		HasNegativeWeightCycle(G)	5044
		AllPairsShortestPaths(G : -)	5044
		MinimumWeightTree(u : -)	5044
		<b>150.13 Bibliography . . . . . 5046</b>	



# Chapter 150

## MULTIGRAPHS

### 150.1 Introduction

*Multigraphs* and *multidigraphs* are graphs and digraphs which may have multiple (i.e., parallel) edges and loops. This is in contrast to simple graphs (see Section 149.1 in the chapter on graphs). In the context of this chapter we use the term “graph” as a generic term for the general vertex-edge incident structure, and the term “multigraph” as a generic term whenever we want to emphasize the possible existence of multiple edges and loops. Thus, graphs and multigraphs may be directed or undirected. The meaning of the terms “graph” and “multigraph” should be made clear from the context in which they occur.

Multigraphs are represented in the form of an adjacency list; for more information on this topic we refer the reader to Section 149.3. As is the case for simple graphs, the vertices and edges of a multigraph may be given “decorations”. More precisely, they may be labelled; in addition the edges may also be assigned a capacity (if one is interested in flow problems) and/or a weight (for investigation of shortest path problems).

For convenience, MAGMA provides users with *networks*: in the MAGMA context, a network is understood as being a multidigraph whose edges are *always* given a capacity. Any MAGMA function that takes a network as an argument will usually take any graph whose edges have a capacity as an argument. Networks are covered in Chapter 151; a few functionalities specific to networks will, however, be discussed in the general multigraph context whenever there is a need to highlight some differences between networks and general multi(di)graphs.

In particular, almost all the standard graph construction functions (see Section 150.5) preserve the graph’s support set and vertex and edge decorations. That is, the resulting graph will have a support and vertex and edge decorations compatible with the original graph and the operation performed on that graph.

A MAGMA multigraph object has type `GrphMultUnd`, a multidigraph has type `GrphMultDir`, and a network has type `GrphNet`. All three objects are of type `GrphMult`.

The multigraph facilities represent a significant subset of the (simple) graph functionality. Consequently, some sections of this chapter are very similar to their counterparts in Chapter 149. For the sake of clarity, this chapter describes the complete multigraph functionality.

## 150.2 Construction of Multigraphs

In this implementation, the order  $n$  of a multigraph or multidigraph is bounded by 134217722. See Section 149.2.1 in Chapter 149 for more details.

### 150.2.1 Construction of a General Multigraph

Undirected multigraphs are constructed in a similar way to graphs (Subsection 149.2.2).

<code>MultiGraph&lt; n   edges &gt;</code>
<code>MultiGraph&lt; S   edges &gt;</code>

Construct the multigraph  $G$  with vertex-set  $V = \{@v_1, v_2, \dots, v_n@\}$  (where  $v_i = i$  for each  $i$  if the first form of the constructor is used, or the  $i$ th element of the enumerated or indexed set  $S$  otherwise), and edge-set  $E = \{e_1, e_2, \dots, e_q\}$ . This function returns three values: The multigraph  $G$ , the vertex-set  $V$  of  $G$ ; and the edge-set  $E$  of  $G$ .

The elements of  $E$  are specified by the list *edges*, where the items of *edges* may be objects of the following types:

- (a) A pair  $\{v_i, v_j\}$  of vertices in  $V$ . The undirected edge  $\{v_i, v_j\}$  from  $v_i$  to  $v_j$  will be added to the edge-set for  $G$ .
- (b) A tuple of the form  $\langle v_i, N_i \rangle$  where  $N_i$  will be interpreted as a set of neighbours for the vertex  $v_i$ . The elements of the sets  $N_i$  must be elements of  $V$ . If  $N_i = \{u_1, u_2, \dots, u_r\}$ , the edges  $\{v_i, u_1\}, \dots, \{v_i, u_r\}$  will be added to  $G$ .
- (c) A sequence  $[N_1, N_2, \dots, N_n]$  of  $n$  sets, where  $N_i$  will be interpreted as a set of neighbours for the vertex  $v_i$ . The edges  $\{v_i, u_i\}$ ,  $u_i \in N_i$ , are added to  $G$ .

In addition to these three basic ways of specifying the *edges* list, the items in *edges* may also be:

- (d) An edge  $e$  of a graph or digraph or multigraph or multidigraph or network of order  $n$ . If  $e$  is an edge from  $u$  to  $v$ , then the edge  $\{u, v\}$  is added to  $G$ .
- (e) An edge-set  $E$  of a graph or digraph or multigraph or multidigraph or network of order  $n$ . Every edge  $e$  in  $E$  will be added to  $G$  according to the rule set out for a single edge.
- (f) A graph or a digraph or a multigraph or a multidigraph or a network  $H$  of order  $n$ . Every edge  $e$  in  $H$ 's edge-set is added to  $G$  according to the rule set out for a single edge.
- (g) A set of
  - (i) Pairs of the form  $\{v_i, v_j\}$  of vertices in  $V$ .
  - (ii) Tuples of the form  $\langle v_i, N_i \rangle$  where  $N_i$  will be interpreted as a set of neighbours for the vertex  $v_i$ .
  - (iii) Edges of a graph or digraph or multigraph or multidigraph or network of order  $n$ .



- (iv) Graphs or digraphs or multigraphs or multidigraphs or networks of order  $n$ .
- (h) A sequence of
  - (i) Tuples of the form  $\langle v_i, N_i \rangle$  where  $N_i$  will be interpreted as a set of neighbours for the vertex  $v_i$ .

**Example H150E1**


---

```
> G := MultiGraph< 3 | < 1, {2, 3} >, < 1, {2} >, < 2, {2, 3} > >;
> G;
Multigraph
Vertex Neighbours
1      2 3 2 ;
2      3 2 2 1 1 ;
3      2 1 ;
```

---

**150.2.2 Construction of a General Multidigraph**

Multidigraphs are constructed in the same way as digraphs (Subsection 149.2.3).

MultiDigraph< n   edges >
---------------------------

MultiDigraph< S   edges >
---------------------------

Construct the multidigraph  $G$  with vertex-set  $V = \{@v_1, v_2, \dots, v_n@\}$  (where  $v_i = i$  for each  $i$  if the first form of the constructor is used, or the  $i$ th element of the enumerated or indexed set  $S$  otherwise), and edge-set  $E = \{e_1, e_2, \dots, e_q\}$ . This function returns three values: The multidigraph  $G$ , the vertex-set  $V$  of  $G$ ; and the edge-set  $E$  of  $G$ .

The elements of  $E$  are specified by the list *edges*, where the items of *edges* may be objects of the following types:

- (a) A pair  $[v_i, v_j]$  of vertices in  $V$ . The directed edge  $[v_i, v_j]$  from  $v_i$  to  $v_j$  will be added to the edge-set for  $G$ .
- (b) A tuple of the form  $\langle v_i, N_i \rangle$  where  $N_i$  will be interpreted as a set of out-neighbours for the vertex  $v_i$ . The elements of the sets  $N_i$  must be elements of  $V$ . If  $N_i = \{u_1, u_2, \dots, u_r\}$ , the edges  $[v_i, u_1], \dots, [v_i, u_r]$  will be added to  $G$ .
- (c) A sequence  $[N_1, N_2, \dots, N_n]$  of  $n$  sets, where  $N_i$  will be interpreted as a set of out-neighbours for the vertex  $v_i$ . All the edges  $[v_i, u_i]$ ,  $u_i \in N_i$ , are added to  $G$ .

In addition to these four basic ways of specifying the *edges* list, the items in *edges* may also be:

- (d) An edge  $e$  of a graph or digraph or multigraph or multidigraph or network of order  $n$ . If  $e$  is an edge from  $u$  to  $v$ , then the edge  $[u, v]$  is added to  $G$ . Thus, if  $e$  is an undirected edge from  $u$  to  $v$ , both edges  $[u, v]$  and  $[v, u]$  are added to  $G$ .
- (e) An edge-set  $E$  of a graph or digraph or multigraph or multidigraph or network of order  $n$ . Every edge  $e$  in  $E$  will be added to  $G$  according to the rule set out for a single edge.
- (f) A graph or a digraph or a multigraph or a multidigraph or a network  $H$  of order  $n$ . Every edge  $e$  in  $H$ 's edge-set is added to  $G$  according to the rule set out for a single edge.
- (g) A set of
  - (i) Pairs of the form  $[v_i, v_j]$  of vertices in  $V$ .
  - (ii) Tuples of the form  $\langle v_i, N_i \rangle$  where  $N_i$  will be interpreted as a set of out-neighbours for the vertex  $v_i$ .
  - (iii) Edges of a graph or digraph or multigraph or multidigraph or network of order  $n$ .
  - (iv) Graphs or digraphs or multigraphs or multidigraphs or networks of order  $n$ .
- (h) A sequence of
  - (i) Tuples of the form  $\langle v_i, N_i \rangle$  where  $N_i$  will be interpreted as a set of out-neighbours for the vertex  $v_i$ .

---

**Example H150E2**

```
> G := MultiDigraph< 3 | < 1, {2, 3} >, < 1, {2} >, < 2, {2, 3} > >;
> G;
Multidigraph
Vertex  Neighbours
1       2 3 2 ;
2       3 2 ;
3       ;
```

---

### 150.2.3 Printing of a Multi(di)graph

A multi(di)graph is displayed by listing, for each vertex, all of its adjacent vertices. If the multigraph has multiple edges from  $u$  to  $v$ , then the adjacency list of  $u$  contains as many copies of the vertex  $v$  as there are edges from  $u$  to  $v$ .

The vertices in the adjacency list are not ordered, they appear in the order in which they were created. See the previous examples H150E1 and H150E2.

### 150.2.4 Operations on the Support

The support of a multi(di)graph is subject to exactly the same operations as simple graphs (see Subsection 149.2.4).

Support( $G$ )
----------------

Support( $V$ )
----------------

The indexed set used in the construction of  $G$  (or the graph for which  $V$  is the vertex-set), or the standard set  $\{@1, \dots, n@\}$  if it was not given.

ChangeSupport( $G, S$ )
-------------------------

If  $G$  is a graph having  $n$  vertices and  $S$  is an indexed set of cardinality  $n$ , return a new graph  $H$  equal to  $G$  but whose support is  $S$ . That is,  $H$  is structurally equal to  $G$  and its vertex and edge decorations are the *same* as those for  $G$  (see Sections 150.4.1 and 150.4.2).

ChangeSupport( $\sim G, S$ )
------------------------------

The procedural version of the above function.

StandardGraph( $G$ )
----------------------

Returns a graph  $H$  that is isomorphic to  $G$  but defined on the standard support. That is,  $H$  is structurally equal to  $G$  and its vertex and edge decorations are the same as those for  $G$ .

#### Example H150E3

---

```
> S := {@ "a", "b", "c" @};
> G := MultiGraph< S | < 1, {2, 3} >, < 1, {2} >, < 2, {2, 3} > >;
> G;
Multigraph
Vertex  Neighbours
c      b a b ;
b      a b b c c ;
a      b c ;
> StandardGraph(G);
Multigraph
Vertex  Neighbours
1      2 3 2 ;
2      3 2 2 1 1 ;
3      2 1 ;
```

---

### 150.3 The Vertex–Set and Edge–Set of Multigraphs

Much of the functionality for simple graphs (see Section 149.4) also applies to multigraphs. We will not repeat here the functions pertaining to the vertex-set and edge-set of a graph, but concentrate instead on the edges. Indeed, there is a difference in the manner in which multigraph edges are created and accessed when compared to simple graph edges.

Since multigraphs may have multiple edges from  $u$  to  $v$ , it is necessary to know how many of these edges there are, and how each can be accessed. More importantly a scheme is required to uniquely identify an edge.

Recall that a multigraph is represented by means of an adjacency list, so any edge from  $u$  to  $v$  can be identified by its index in this list. If there is more than one edge from  $u$  to  $v$ , then a list of indices will identify each edge from  $u$  to  $v$ . Thus, the index of the edge in the adjacency list will be the means by which this edge can be *uniquely* identified.

This has a bearing on how an edge can be coerced into a multigraph: Successful coercion will require two vertices  $u$  and  $v$  so that  $v$  is a neighbour of  $u$ , and a valid index  $i$  in the multigraph's adjacency list. That is, the position  $i$  in the list is the index of an edge from  $u$  to  $v$ .

EdgeIndices( $u$ ,  $v$ )

Indices( $u$ ,  $v$ )

Given vertices  $u$  and  $v$  of a multigraph  $G$ , returns the indices of the possibly multiple edge from  $u$  to  $v$ .

EdgeMultiplicity( $u$ ,  $v$ )

Multiplicity( $u$ ,  $v$ )

Given vertices  $u$  and  $v$  of a multigraph  $G$ , returns the multiplicity of the possibly multiple edge from  $u$  to  $v$ . Returns 0 if  $u$  is not adjacent to  $v$ .

Edges( $u$ ,  $v$ )

Given vertices  $u$  and  $v$  of a multigraph  $G$ , returns all the edges from  $u$  to  $v$  as a sequence of elements of the edge-set of  $G$ .

IncidentEdges( $u$ )

Given a vertex  $u$  of an undirected multigraph  $G$ , returns all the edges incident to  $u$  as a set of elements of the edge-set of  $G$ . If  $G$  is a multidigraph, the function returns all the edges incident to *and* from  $u$  as a set of elements of the edge-set of  $G$ .

$E ! < \{ u, v \}, i >$

Given the edge-set  $E$  of the undirected multigraph  $G$  and objects  $u, v$  belonging to the support of  $G$  corresponding to adjacent vertices, returns the edge from  $u$  to  $v$  which corresponds to the edge with index  $i$  in the adjacency list of  $G$ . This requires that the edge at  $i$  in the adjacency list of  $G$  is an edge from  $u$  to  $v$ .

$E ! < [u, v], i >$

Given the edge-set  $E$  of the multidigraph  $G$  and objects  $u, v$  belonging to the support of  $G$  corresponding to adjacent vertices, returns the edge from  $u$  to  $v$  which corresponds to the edge with index  $i$  in the adjacency list of  $G$ . This requires that the edge at  $i$  in the adjacency list of  $G$  is an edge from  $u$  to  $v$ .

$E . i$

Let  $E$  be the edge-set of  $G$ . If  $G$  is a simple graph (digraph) then this function is as described in Subsection 149.4.2.

If  $G$  is a multigraph or multidigraph, then this function returns the edge at index  $i$  in the adjacency list of  $G$ , provided  $i$  is a valid index.

$\text{EndVertices}(e)$

Given an edge  $e$  in an undirected multigraph  $G$ , returns the end vertices of  $e$  as a set of vertices  $\{u, v\}$ . If  $G$  is a multidigraph, returns  $e$ 's end vertices as a sequence of vertices  $[u, v]$ .

$\text{InitialVertex}(e)$

Given an edge  $e = \{u, v\}$  or  $e = [u, v]$ , returns vertex  $u$ . This is useful in the undirected case since it indicates, where relevant, the direction in which the edge has been traversed.

$\text{TerminalVertex}(e)$

Given an edge  $e = \{u, v\}$  or  $e = [u, v]$ , returns vertex  $v$ . This is useful in the undirected case since it indicates, where relevant, the direction in which the edge has been traversed.

$\text{Index}(e)$

Given an edge  $e$  in a multi(di)graph  $G$ , returns the index of  $e$  in the adjacency list of  $G$ .

$s \text{ eq } t$

Returns **true** if the edge  $s$  is equal to the edge  $t$ . If  $s$  and  $t$  are edges in a multi(di)graph  $G$ , returns **true** if and only if  $s$  and  $t$  have the same index in the adjacency list of  $G$ .

**Example H150E4**

---

```
> G := MultiGraph< 3 | < 1, {2, 3} >, < 1, {2} >, < 2, {2, 3} > >;
```

The graph  $G$  has a loop at vertex 2:

```
> Edges(G);
{@ < {1, 2}, 1 >, < {1, 2}, 5 >, < {1, 3}, 3 >, < {2, 2}, 7 >, < {2,
3}, 9 > @}
> E := EdgeSet(G);
> E.7;
< {2, 2}, 7 >
> assert InitialVertex(E.7) eq TerminalVertex(E.7);
```

The graph  $G$  has a multiple edge from vertex 1 to vertex 2:

```
> u := VertexSet(G)!1;
> v := VertexSet(G)!2;
> I := EdgeIndices(u, v);
> I;
[ 5, 1 ]
> assert #I eq Multiplicity(u, v);
>
> E := EdgeSet(G);
> e1 := E!< { 1, 2 }, I[1] >;
> e2 := E!< { 1, 2 }, I[2] >;
> e1, e2;
< {1, 2}, 5 > < {1, 2}, 1 >
> EndVertices(e1);
{ 1, 2 }
> EndVertices(e2);
{ 1, 2 }
> assert Index(e1) eq I[1];
> assert Index(e2) eq I[2];
>
> assert e1 eq E.I[1];
> assert not e2 eq E.I[1];
> assert e2 eq E.I[2];
> assert not e1 eq E.I[2];
```

We have seen that these two edges can be accessed directly:

```
> Edges(u, v);
[ < {1, 2}, 5 >, < {1, 2}, 1 > ]
```

---

## 150.4 Vertex and Edge Decorations

### 150.4.1 Vertex Decorations: Labels

In this implementation, it is only possible to assign labels as vertex decorations.

A vertex labelling of a graph  $G$  is a partial map from the vertex-set of  $G$  into a set  $L$  of labels.

**AssignLabel( $\sim G$ ,  $u$ ,  $l$ )**

Assigns the label  $l$  to the vertex  $u$  in the graph  $G$ .

**AssignLabels( $\sim G$ ,  $S$ ,  $L$ )**

Assigns the labels in  $L$  to the corresponding vertices in  $G$  in the sequence or indexed set  $S$ . If for vertex  $u$  the corresponding entry in  $L$  is not defined then any existing label of  $u$  is removed.

**AssignVertexLabels( $\sim G$ ,  $L$ )**

Assigns the labels in  $L$  to the corresponding vertices in the graph  $G$ .

**IsLabelled( $u$ )**

Returns **true** if and only if the vertex  $u$  has a label.

**IsLabelled( $V$ )**

Returns **true** if and only if the vertex-set  $V$  is labelled.

**IsVertexLabelled( $G$ )**

Returns **true** if and only if vertices of the graph  $G$  are labelled.

**Label( $u$ )**

The label of the vertex  $u$ . An error is raised if  $u$  is not labelled.

**Labels( $S$ )**

The sequence  $L$  of labels of the vertices in the sequence  $S$ . If an element of  $S$  has no label, then the corresponding entry in  $L$  is undefined.

**Labels( $V$ )**

The sequence  $L$  of labels of the vertices in the vertex-set  $V$ . If an element of  $V$  has no label, then the corresponding entry in  $L$  is undefined.

**VertexLabels( $G$ )**

The sequence  $L$  of labels of the vertices of  $G$ . If a vertex of  $G$  has no label, then the corresponding entry in  $L$  is undefined.

**DeleteLabel( $\sim G$ ,  $u$ )**

Removes the label of the vertex  $u$ .

Remove the labels of the vertices in  $S$ .

<code>DeleteVertexLabels(~G)</code>
-------------------------------------

Remove the labels of the vertices in the graph  $G$ .

### 150.4.2 Edge Decorations

Simple graph or multigraph edges can be assigned three different types of decorations:

- a label,
- a capacity,
- a weight.

Edge labels can be objects of any MAGMA type. Edge capacities must be non-negative integers (and loops must be assigned a zero capacity). Edge weights must be elements of a totally ordered ring. Not all edges need to be assigned labels when labelling edges, nor do all edges need to be assigned capacities or weights when assigning either decoration. In the latter case, if some edges have been assigned a capacity or a weight, then the capacity or weight of any remaining unassigned edge is always taken to be zero.

One may want to assign capacities to edges in order to apply a network-flow algorithm to the graph (Section 151.4). By assigning weights to edges one may also be able to run a shortest-path algorithm (Section 150.12).

#### 150.4.2.1 Assigning Edge Decorations

<code>AssignLabel(~G, e, l)</code>
------------------------------------

<code>AssignCapacity(~G, e, c)</code>
---------------------------------------

<code>AssignWeight(~G, e, w)</code>
-------------------------------------

Assigns the label  $l$  or the capacity  $c$  or the weight  $w$  to the edge  $e$  in the graph  $G$ . The capacity of any edge must be a non-negative integer except in the case of a loop when it must be zero. The weight  $w$  must be an element from a totally ordered ring.

<code>AssignLabels(~G, S, D)</code>
-------------------------------------

<code>AssignCapacities(~G, S, D)</code>
---

<code>AssignWeights(~G, S, D)</code>
--------------------------------------

Assigns the labels or capacities or weights in the sequence  $D$  to the corresponding edges in the sequence or indexed set  $S$ . If for some edge  $e$  the corresponding entry in  $D$  is not defined then any existing label or capacity or weight of  $e$  is removed. The same constraints regarding capacity and weight apply as in the single edge assignment case.



<code>AssignEdgeLabels(<math>\sim G</math>, <math>D</math>)</code>
--

<code>AssignCapacities(<math>\sim G</math>, <math>D</math>)</code>
--

<code>AssignWeights(<math>\sim G</math>, <math>D</math>)</code>
---

Assigns the labels or capacities or weights in the sequence  $D$  to the corresponding edges in graph  $G$ . Let  $E$  be the edge-set of  $G$  and let  $d$  be the decoration at position  $i$  in  $D$ , that is,  $d = D[i]$ . Then the corresponding edge  $e$  in  $E$  which will be decorated is  $E.i$  (see E.i).

If for some edge  $e$  the corresponding entry in  $D$  is not defined then any existing label or capacity or weight of  $e$  is removed. The same constraints regarding capacity and weight apply as in the single edge assignment case.

### 150.4.2.2 Testing for Edge Decorations

While it is the case that an edge is considered to be labelled if and only if it has been assigned a label, an edge may have a default capacity (weight) of zero. Let  $e$  be an edge of a graph  $G$  and assume that  $e$  has not been assigned a capacity (weight).

If any other edge of  $G$  has been assigned a capacity (weight), then the edge-set of  $G$  is considered to be capacitated (weighted). In which case the capacity (weight) of  $e$  has a default value of zero.

If no other edge of  $G$  has been assigned a capacity (weight), then the edge-set of  $G$  is considered to be uncapacitated (unweighted). In which case asking for the capacity (weight) of  $e$  results in an error.

This is in contrast to the labelling situation where there is no concept of a “default” label. The edge-set of  $G$  is considered to be labelled if and only if at least one edge of  $G$  has been labelled, while any edge of  $G$  is considered to be labelled if and only if  $e$  has been labelled.

<code>IsLabelled(<math>e</math>)</code>
---

Returns **true** if and only if the edge  $e$  has a label.

<code>IsLabelled(<math>E</math>)</code>
---

Returns **true** if and only if the edge-set is labelled; that is, if and only if at least one edge of  $E$  has been assigned a label.

<code>IsEdgeLabelled(<math>G</math>)</code>
---

Returns **true** if and only if the edge-set of  $G$  is labelled.

<code>IsCapacitated(<math>E</math>)</code>
--

Returns **true** if and only if the edge-set is capacitated; that is, if and only if at least one edge of  $E$  has been assigned a capacity.

<code>IsEdgeCapacitated(<math>G</math>)</code>
--

Returns **true** if and only if the edge-set of  $G$  is capacitated.

**IsWeighted(*E*)**

Returns **true** if and only if the edge-set is weighted; that is, if and only if at least one edge of  $E$  has been assigned a weight.

**IsEdgeWeighted(*G*)**

Returns **true** if and only if the edge-set of  $G$  is weighted.

### 150.4.2.3 Reading Edge Decorations

An edge may have a default capacity and weight of zero, see Subsubsection 150.4.2.2 for more details.

**Label(*e*)**

The label of the edge  $e$ . An error is raised if  $e$  has not been assigned a capacity.

**Capacity(*e*)**

The capacity of the edge  $e$ . Let  $G$  be the parent graph of  $e$ . An error is raised if the edge-set of  $G$  is uncapacitated. If the edge-set of  $G$  is capacitated but  $e$  has not been assigned a capacity, then **Capacity(*e*)** returns zero as the default value.

**Weight(*e*)**

The weight of the edge  $e$ . Let  $G$  be the parent graph of  $e$ . An error is raised if the edge-set of  $G$  is unweighted. If the edge-set of  $G$  is weighted but  $e$  has not been assigned a weight, then **Weight(*e*)** returns zero as the default value.

**Labels(*S*)**

**Capacities(*S*)**

**Weights(*S*)**

The sequence  $D$  of labels or capacities or weights of the edges in the sequence  $S$ . Let  $E$  be the edge-set of the parent graph of the edges. If  $E$  is unlabelled or uncapacitated or unweighted then  $D$  is the null sequence. If an element of  $S$  has no label then the corresponding entry in  $D$  is undefined. If an element of  $S$  has no capacity or weight while  $E$  is capacitated or weighted then the corresponding entry in  $D$  has the default value of zero.

**Labels(*E*)**

**Capacities(*E*)**

**Weights(*E*)**

The sequence  $D$  of labels or capacities or weights of the edges in the edge-set  $E$ . If  $E$  is unlabelled or uncapacitated or unweighted then  $D$  is the null sequence. If an element of  $E$  has no label then the corresponding entry in  $D$  is undefined. If an element of  $E$  has no capacity or weight while  $E$  is capacitated or weighted then the corresponding entry in  $D$  has the default value of zero. The corresponding entry  $i$  in  $D$  of any edge  $e$  is such that  $e = E.i$  (see **E.i**).

EdgeLabels( $G$ )
-------------------

EdgeCapacities( $G$ )
-----------------------

EdgeWeights( $G$ )
--------------------

The sequence  $D$  of labels or capacities or weights of the edges in edge-set  $E$  of the graph  $G$ . If  $E$  is unlabelled or uncapacitated or unweighted then  $D$  is the null sequence. If an element of  $E$  has no label then the corresponding entry in  $D$  is undefined. If an element of  $E$  has no capacity or weight while  $E$  is capacitated or weighted then the corresponding entry in  $D$  has the default value of zero. The corresponding entry  $i$  in  $D$  of any edge  $e$  is such that  $e = E.i$  (see E.i).

#### 150.4.2.4 Deleting Edge Decorations

DeleteLabel( $\sim G$ , $e$ )
-------------------------------

DeleteCapacity( $\sim G$ , $e$ )
----------------------------------

DeleteWeight( $\sim G$ , $e$ )
--------------------------------

Removes the label or capacity or weight of the edge  $e$  in the graph  $G$ .

DeleteLabels( $\sim G$ , $S$ )
--------------------------------

DeleteCapacities( $\sim G$ , $S$ )
------------------------------------

DeleteWeights( $\sim G$ , $S$ )
---------------------------------

Remove the labels or capacities or weights of the edges (of the graph  $G$ ) in  $S$ .

DeleteEdgeLabels( $\sim G$ )
------------------------------

DeleteCapacities( $\sim G$ )
------------------------------

DeleteWeights( $\sim G$ )
---------------------------

Remove the labels or capacities or weights of the edges in the edge set of the graph  $G$ .

#### 150.4.3 Unlabelled, or Uncapacitated, or Unweighted Graphs

Starting with a graph  $G$ , the functions below return a graph that is isomorphic as a simple graph to  $G$ , but without the vertex and edge labels of  $G$ , (or without the edge capacities or weights of  $G$  in the case of a network). Should one require a copy of a graph without the support of  $G$ , see Subsection 149.2.4. Should one require a copy of a graph without the support of  $G$  and without the vertex/edge decorations of  $G$ , see Subsection 150.6.3.

UnlabelledGraph( $G$ )
------------------------

Return the (vertex and edge) unlabelled graph structurally identical to  $G$ , whose edges have the same capacities and weights as those in  $G$ . The support of  $G$  is also retained in the resulting graph.

**UncapacitatedGraph(G)**

Return the uncapacitated graph structurally identical to  $G$ , whose vertices and edges have the same labels, and whose edges have the same weights as those in  $G$ . The support of  $G$  is also retained in the resulting graph.

**UnweightedGraph(G)**

Return the unweighted graph structurally identical to  $G$ , whose vertices and edges have the same labels, and whose edges have the same capacities as those in  $G$ . The support of  $G$  is also retained in the resulting graph.

**Example H150E5**

The labelling operations are illustrated by constructing a 2-colouring of the complete bipartite graph  $K_{3,4}$ . Use is made of the function `Distance( $u, v$ )` which returns the distance between vertices  $u$  and  $v$ .

```
> K34, V, E := BipartiteGraph(3, 4);
> L := [ IsEven(Distance(V!1, v)) select "red" else "blue" : v in Vertices(K34) ];
> AssignLabels(Vertices(K34), L);
> VertexLabels(K34);
[ red, red, red, blue, blue, blue, blue ]
```

**Example H150E6**

Another illustration is the creation of the Cayley graph of a group. In this example `Sym(4)` is used.

```
> G<a,b> := FPGGroup(Sym(4));
> I, m := Transversal(G, sub<G | 1>);
> S := Setseq(Generators(G));
> N := [ {m(a*b) : b in S} : a in I ];
> graph := StandardGraph(Digraph< I | N >);
> AssignLabels(VertexSet(graph), IndexedSetToSequence(I));
> V := VertexSet(graph);
> E := EdgeSet(graph);
> for i in [1..#I] do
>   AssignLabels([ E![V | i, Index(I, m(I[i]*s))] : s in S ], S);
> end for;
```

In this graph,  $[1,2,5,4]$  is a cycle. So the corresponding edge labels should multiply to the identity.

```
> &*Labels([ EdgeSet(graph) | [1,2], [2,5], [5,4], [4,1] ]);
a^4
> G;
```

Finitely presented group G on 2 generators

Relations

```
b^2 = Id(G)
a^4 = Id(G)
(a^-1 * b)^3 = Id(G)
```

**Example H150E7**

---

We turn to multigraphs to illustrate a point with respect to the listing of the edge decorations. We assign random labels, capacities and weights to a multigraph.

```
> G := MultiGraph< 3 | < 1, {2, 3} >, < 1, {2} >, < 2, {2, 3} > >;
> E := EdgeSet(G);
> I := Indices(G);
>
> for i in I do
>   AssignLabel(~G, E.i, Random([ "a", "b", "c", "d" ]));
>   if not InitialVertex(E.i) eq TerminalVertex(E.i) then
>     AssignCapacity(~G, E.i, Random(1, 3));
>   end if;
>   AssignWeight(~G, E.i, Random(1, 3));
> end for;
>
> EdgeLabels(G);
[ c, undef, c, undef, d, undef, c, undef, c ]
> EdgeCapacities(G);
[ 2, undef, 3, undef, 2, undef, 0, undef, 3 ]
> EdgeWeights(G);
[ 3, undef, 2, undef, 1, undef, 2, undef, 1 ]
```

Since  $G$  is undirected, the edge  $\{v, u\}$  and the edge  $\{u, v\}$  are the same object, and thus they have the same index:

```
> V := VertexSet(G);
> u := V!1;
> v := V!3;
> Indices(u, v);
[ 3 ]
> Indices(v, u);
[ 3 ]
```

However, since  $G$  is represented by means of an adjacency list, the undirected edge  $\{u, v\}$  is stored twice in the list, and so there are two positions in the list associated with the edge. By convention, these positions are contiguous, but, more importantly from the user's perspective, the function `Index` that returns the index of the edge  $\{u, v\}$  *always* returns the odd index associated with the edge.

This explains why, for an undirected multigraph, the sequence returned by a function like `EdgeLabels` will always have undefined elements.

Finally note that the loop  $\{2, 2\}$ , which was assigned no capacity, is shown to have capacity zero:

```
> E := EdgeSet(G);
> E.7;
< {2, 2}, 7 >
> Capacity(E.7);
0
```

This is in accordance with the definition of the default value for edge capacity and weight (see Subsubsection 150.4.2.2).

## 150.5 Standard Construction for Multigraphs

As noted in the Introduction 150.1, most of the functions listed in this section correctly handle a graph's support and vertex/edge decorations. That is, these attributes are inherited by the graph created as a result of applying such a function.

### 150.5.1 Subgraphs

The construction of subgraphs from multigraphs or multidigraphs is similar to the construction of subgraphs from simple graphs or digraphs (see Subsection 149.6.1).

Note that the support set, vertex labels and edge decorations are transferred from the supergraph to the subgraph.

sub< G   list >
-----------------

Construct the multigraph  $H$  as a subgraph of  $G$ . The function returns three values: The multigraph  $H$ , the vertex-set  $V$  of  $H$ ; and the edge-set  $E$  of  $H$ . If  $G$  has a support set and/or if  $G$  has vertex/edge labels, and/or edge capacities or edge weights then *all* these attributes are transferred to the subgraph  $H$ .

The elements of  $V$  and of  $E$  are specified by the list *list* whose items can be objects of the following types:

- (a) A vertex of  $G$ . The resulting subgraph will be the subgraph induced on the subset of  $\text{VertexSet}(G)$  defined by the vertices in *list*.
- (b) An edge of  $G$ . The resulting subgraph will be the subgraph with vertex-set  $\text{VertexSet}(G)$  whose edge-set consists of the edges in *list*.
- (c) A set of
  - (i) Vertices of  $G$ .
  - (ii) Edges of  $G$ .

If the list of vertices and edges happens to contain duplicate elements, they will be ignored by the subgraph constructor. It is easy to recover the map that sends the vertices of the subgraph to the vertices of the supergraph and vice-versa: Simply coerce the vertex of the subgraph into the supergraph's vertex-set, and, if applicable, coerce the vertex of the supergraph into the subgraph's vertex-set.

**Example H150E8**

---

We create a multidigraph  $G$  with a support set; we assign labels to its vertices, and labels, capacities and weights to its edges. This demonstrates the fact that any subgraph of  $G$  retains the support set, as well as the vertex and edge decorations.

```

> S := {@ "a", "b", "c" @};
> G := MultiDigraph< S | < 1, {2, 3} >, < 1, {2} >, < 2, {2, 3} > >;
> G;
Multidigraph
Vertex  Neighbours
a      b c b ;
b      c b ;
c      ;
>
> V := VertexSet(G);
> for u in V do
>   AssignLabel(~G, u, Random([ "X", "Y", "Z" ]));
> end for;
>
> E := EdgeSet(G);
> I := Indices(G);
> for i in I do
>   AssignLabel(~G, E.i, Random([ "D", "E", "F" ]));
>   if not InitialVertex(E.i) eq TerminalVertex(E.i) then
>     AssignCapacity(~G, E.i, Random(1, 3));
>   end if;
>   AssignWeight(~G, E.i, Random(1, 3));
> end for;
>
> VertexLabels(G);
[ Z, Y, Z ]
> EdgeLabels(G);
[ E, F, D, E, E ]
> EdgeCapacities(G);
[ 2, 1, 3, 0, 2 ]
> EdgeWeights(G);
[ 2, 1, 3, 1, 1 ]
>
> V := VertexSet(G);
> H := sub< G | V!1, V!2 >;
> H;
Multidigraph
Vertex  Neighbours
a      b b ;
b      b ;
>
> for u in VertexSet(H) do
>   assert Label(u) eq Label(V!u);

```

```

> end for;
>
> for e in EdgeSet(H) do
>   u := InitialVertex(e);
>   v := TerminalVertex(e);
>
>   assert SequenceToSet(Labels(Edges(u, v)))
>   eq SequenceToSet(Labels(Edges(V!u, V!v)));
>   assert SequenceToSet(Capacities(Edges(u, v)))
>   eq SequenceToSet(Capacities(Edges(V!u, V!v)));
>   assert SequenceToSet(Weights(Edges(u, v)))
>   eq SequenceToSet(Weights(Edges(V!u, V!v)));
> end for;

```

Note that since  $G$  is a multidigraph it is not possible to coerce an edge of  $H$  into the edge-set of  $G$ . This is because edge coercion for multi(di)graphs involves the coercion of both the end-vertices *and* the index of the edge.

A correspondence is established between the edges of  $H$  and the edges of  $G$  by retrieving *all* the edges in  $H$  and  $G$  having same end-vertices.

---

## 150.5.2 Incremental Construction of Multigraphs

The complete functionality for adding and removing vertices or edges in simple graphs (see Subsection 149.6.2) is also available for multigraphs.

Note that some functions adding an edge to a multigraph also return the newly created edge. This feature is useful when there is a need to determine the index of this edge in the adjacency list of the modified multigraph. Also, there is the possibility of removing *all* the edges from  $u$  to  $v$  for given vertices  $u$  and  $v$  of the multigraphs.

Further, whenever a vertex or an edge is added or removed from a graph, the existing vertex labels and the edge labels or capacities or weights are retained. The support of the graph is retained in all cases except when adding a new vertex.

Unless otherwise specified, each of the functions described in this section returns three values:

- (i) The multigraph  $G$ ;
- (ii) The vertex-set  $V$  of  $G$ ;
- (iii) The edge-set  $E$  of  $G$ .

### 150.5.2.1 Adding Vertices

$G + n$

Given a non-negative integer  $n$ , adds  $n$  new vertices to the multigraph  $G$ . The existing vertex labels and edge labels or capacities or weights are retained, but the support will become the standard support.



$G \mathrel{+:=} n$
---------------------

<code>AddVertex(<math>\sim G</math>)</code>
---

<code>AddVertices(<math>\sim G</math>, <math>n</math>)</code>
---

The procedural version of the previous function. `AddVertex` adds one vertex only to  $G$ .

<code>AddVertex(<math>\sim G</math>, <math>l</math>)</code>
---

Given a graph  $G$  and a label  $l$ , adds a new vertex with label  $l$  to  $G$ . The existing vertex labels and edge labels or capacities or weights are retained, but the support will become the standard support.

<code>AddVertices(<math>\sim G</math>, <math>n</math>, <math>L</math>)</code>
---

Given a graph  $G$  and a non-negative integer  $n$ , and a sequence  $L$  of  $n$  labels, adds  $n$  new vertices to  $G$  with labels from  $L$ . The existing vertex labels and edge labels or capacities or weights are retained, but the support will become the standard support.

### 150.5.2.2 Removing Vertices

$G - v$
---------

$G - U$
---------

Given a vertex  $v$  of  $G$ , or a set  $U$  of vertices of  $G$ , removes  $v$  (or the vertices in  $U$ ) from  $G$ . The support, vertex labels, and edge labels or capacities or weights are retained.

$G \mathrel{-:=} v$
---------------------

$G \mathrel{-:=} U$
---------------------

<code>RemoveVertex(<math>\sim G</math>, <math>v</math>)</code>
--

<code>RemoveVertices(<math>\sim G</math>, <math>U</math>)</code>
--

The procedural versions of the previous functions.

### 150.5.2.3 Adding Edges

$G + \{ u, v \}$
------------------

$G + [ u, v ]$
----------------

Given a graph  $G$  and a pair of vertices of  $G$ , add the edge to  $G$  described by this pairs. If  $G$  is undirected then the edge must be given as a set of (two) vertices, if  $G$  is directed the edge is given as a sequence of (two) vertices. If  $G$  is a network, then the edge is added with a capacity of 1 (0 if a loop). The support, vertex labels, and edge labels or capacities or weights are retained. This set of functions has two return values: The first is the modified graph and the second is the newly created edge. This feature is especially useful when adding parallel edges.

$$G + \{ \{ u, v \} \}$$

$$G + [ \{ u, v \} ]$$

$$G + \{ [ u, v ] \}$$

$$G + [ [ u, v ] ]$$

Given a graph  $G$  and a set or a sequence of pairs of vertices of  $G$ , add the edges to  $G$  described by these pairs. If  $G$  is undirected then the edges must be given as a set of (two) vertices, if  $G$  is directed the edges are given as a sequence of (two) vertices. If  $G$  is a network, then the edges are added with a capacity of 1 (0 if a loop). The support, vertex labels, and edge labels or capacities or weights are retained.

$$G += \{ u, v \}$$

$$G += [ u, v ]$$

$$G += [ u, v ]$$

$$G += \{ \{ u, v \} \}$$

$$G += [ \{ u, v \} ]$$

$$G += \{ [ u, v ] \}$$

$$G += [ [ u, v ] ]$$

The procedural versions of the previous four functions.

$$\text{AddEdge}(G, u, v)$$

Given a graph  $G$ , two vertices of  $G$   $u$  and  $v$ , returns a new edge between  $u$  and  $v$ . If  $G$  is a network, the edge is added with a capacity of 1 (0 if loop). The support, vertex labels, and edge labels or capacities or weights are retained. This function has two return values: The first is the modified graph and the second is the newly created edge. This feature is especially useful when adding parallel edges.

$$\text{AddEdge}(G, u, v, l)$$

Given a graph  $G$  which is not a network, two vertices of  $G$   $u$  and  $v$ , and a label  $l$ , adds a new edge with label  $l$  between  $u$  and  $v$ . The support, vertex labels, and edge labels or capacities or weights are retained. This function has two return values: The first is the modified graph and the second is the newly created edge.

$$\text{AddEdge}(G, u, v, c)$$

Given a network  $G$ , two vertices of  $G$   $u$  and  $v$ , and a non-negative integer  $c$ , adds a new edge from  $u$  to  $v$  with capacity  $c$ . The support, vertex labels, and edge labels or capacities or weights are retained. This function has two return values: The first is the modified graph and the second is the newly created edge.

AddEdge( $G, u, v, c, l$ )
----------------------------

Given a network  $G$ , two vertices of  $G$   $u$  and  $v$ , a non-negative integer  $c$ , and a label  $l$ , adds a new edge from  $u$  to  $v$  with capacity  $c$  and label  $l$ . If  $G$  is a network, then add a new edge with label  $l$  and capacity  $c$  between  $u$  and  $v$ . The support, vertex labels, and edge labels or capacities or weights are retained. This function has two return values: The first is the modified graph and the second is the newly created edge.

AddEdge( $\sim G, u, v$ )
---------------------------

AddEdge( $\sim G, u, v, l$ )
------------------------------

AddEdge( $\sim G, u, v, c$ )
------------------------------

AddEdge( $\sim G, u, v, c, l$ )
---------------------------------

Procedural versions of the previous functions for adding edges to a graph.

AddEdges( $G, S$ )
--------------------

Given a graph  $G$ , and a sequence or set  $S$  of pairs of vertices of  $G$ , the edges specified in  $S$  are added to  $G$ . The elements of  $S$  must be sets or sequences of two vertices of  $G$ , depending upon whether  $G$  is undirected or directed respectively.

If  $G$  is a network, the edges are added with a capacity of 1 (0 if a loop). The support, vertex labels, and edge labels or capacities or weights are retained.

AddEdges( $G, S, L$ )
-----------------------

Given a graph  $G$ , a sequence  $S$  of pairs of vertices of  $G$ , and a sequence  $L$  of labels of the same length, the edges specified in  $S$  are added to  $G$  with its corresponding label as given in  $L$ . The elements of  $S$  must be sets or sequences of two vertices of  $G$ , depending upon whether  $G$  is undirected or directed respectively. If  $G$  is a network, the edges are added with a capacity of 1 (0 if loop). The support, vertex labels, and edge labels or capacities or weights are retained.

AddEdges( $\sim G, S$ )
-------------------------

AddEdges( $\sim G, S, L$ )
----------------------------

These are procedural versions of the previous functions for adding edges to a graph.

#### 150.5.2.4 Removing Edges

$G - e$
---------

$G - \{ e \}$
---------------

Given an edge  $e$  or a set  $S$  of edges of a multigraph  $G$ , this function creates a graph that corresponds to  $G$  with the edge  $e$  (respectively, set of edges  $S$ ) removed. The resulting multigraph will have vertex-set  $V(G)$  and edge-set  $E(G) \setminus \{e\}$  (respectively,  $E(G) \setminus S$ ). The support, vertex labels and edge labels are retained on the remaining edges.

$G - \{ \{ u, v \} \}$
------------------------

$G - \{ [u, v] \}$
--------------------

Given a graph  $G$  and a set  $S$  of pairs  $\{u, v\}$  or  $[u, v]$ ,  $u, v$  vertices of  $G$ , this function forms the graph having vertex-set  $V(G)$  and edge-set  $E(G) - S$ . That is, the graph returned is the same as  $G$  except that *all* the edges specified by pairs in  $S$  have been removed. An edge is represented as a set if  $G$  is undirected, and as a sequence otherwise. The support, vertex labels and edge labels are retained on the remaining edges.

$G ::= e$
-----------

$G ::= \{ e \}$
-----------------

$G ::= \{ \{ u, v \} \}$
--------------------------

$G ::= \{ [u, v] \}$
----------------------

$\text{RemoveEdge}(\sim G, e)$
--------------------------------

$\text{RemoveEdges}(\sim G, S)$
---------------------------------

$\text{RemoveEdge}(\sim G, u, v)$
-----------------------------------

These operations represent procedural versions of the previous functions. Whenever an edge is represented as a pair  $\{u, v\}$  or  $[u, v]$  of vertices of  $G$ , it is assumed that *all* the edges from  $u$  to  $v$  are to be removed from  $G$ .

### 150.5.3 Vertex Insertion, Contraction

As in the case of simple graphs (see Subsection 149.6.3) it is possible to insert a vertex in a multigraph edge. The new edges thus created will be unlabelled with their capacities and weights set as follows: (These rules apply regardless as to whether the edge-set is capacitated and/or weighted).

Let  $e$  be an edge from  $u$  to  $v$  with capacity  $c$  and weight  $w$  in a multigraph  $G$ . After the insertion of a new vertex  $x$  in  $e$ , the edge  $e$  will be replaced by two edges, one from  $u$  to  $x$  and the other from  $x$  to  $v$ , *both* having capacity  $c$  and weight  $w$ .

These rules apply regardless as to whether the edge-set is capacitated and/or weighted.

The contraction operation can only be applied to a pair of vertices, since contracting a single multigraph edge which might have parallel edges is meaningless. The graph's support and vertex/edges decorations are retained when contracting its edges.

Each of the functions described below returns three values:

- (i) The multigraph  $G$ ;
- (ii) The vertex-set  $V$  of  $G$ ;
- (iii) The edge-set  $E$  of  $G$ .

**InsertVertex(e)**

Given an edge  $e$  of the multigraph  $G$ , this function inserts a new vertex of degree 2 in  $e$ . If appropriate, the two new edges that replace  $e$  will have the same capacity and weight as  $e$ . They will be unlabelled. The vertex labels and the edge decorations of  $G$  are retained, but the resulting graph will have standard support.

**InsertVertex(T)**

Given a set  $T$  of edges belonging to the multigraph  $G$ , this function inserts a vertex of degree 2 in each edge belonging to the set  $T$ .

**Contract(e)**

Given an edge  $e = \{u, v\}$  of the graph  $G$ , form the graph obtained by removing the edge  $e$  and then identifying the vertices  $u$  and  $v$ . New parallel edges and new loops may result from this operation but any new loop will be assigned zero capacity. With the above exception, the edge decorations are retained, as are the support and the vertex labels of  $G$ .

**Contract(u, v)**

Given vertices  $u$  and  $v$  belonging to the multigraph  $G$ , this function returns the multigraph obtained by identifying the vertices  $u$  and  $v$ . New parallel edges and new loops may result from this operation but any new loop will be assigned zero capacity. With the above exception, the edge decorations are retained, as are the support and the vertex labels of  $G$ .

**Contract(S)**

Given a set  $S$  of vertices belonging to the multigraph  $G$ , this function returns the multigraph obtained by identifying all of the vertices in  $S$ .

### 150.5.4 Unions of Multigraphs

Of the union operations available for simple graphs (see Section 149.7) only **Union** and **EdgeUnion** have been implemented for multigraphs. It is straightforward to write MAGMA code for other union functions with multigraphs.

In contrast with the other standard graph constructions, the support, vertex labels and edge decorations are generally *not* handled by the functions listed below. Thus, the resulting graph will always have standard support and no vertex labels nor edge decorations. The one exception occurs in the case of networks, where edge capacities are properly handled.

**Union(G, H)****G join H**

Given multi(di)graphs  $G$  and  $H$  with disjoint vertex sets  $V(G)$  and  $V(H)$  respectively, this function constructs their union, i.e. the multi(di)graph with vertex-set  $V(G) \cup V(H)$ , and edge-set  $E(G) \cup E(H)$ . The resulting multi(di)graph has standard support and no vertex labels nor edge decorations.

Union( $N$ ,  $H$ )

$N$  join  $H$

Given networks  $N$  and  $H$  with disjoint vertex sets  $V(N)$  and  $V(H)$  respectively, construct their union, i.e. the network with vertex-set  $V(N) \cup V(H)$ , and edge-set  $E(N) \cup E(H)$ . The resulting network has standard support and capacities but neither vertex labels, edge labels nor weights retained.

& join  $S$

The union of the multigraphs or networks in the sequence or the set  $S$ .

EdgeUnion( $G$ ,  $H$ )

Given multi(di)graphs  $G$  and  $H$  having the same number of vertices, construct their edge union  $K$ . This construction identifies the  $i$ -th vertex of  $G$  with the  $i$ -th vertex of  $H$  for all  $i$ . The edge union has the same vertex-set as  $G$  (and hence as  $H$ ) and there is an edge from  $u$  to  $v$  in  $K$  if and only if there is an edge from  $u$  to  $v$  in either  $G$  or  $H$ . The resulting multi(di)graph has standard support but neither vertex labels nor edge decorations.

EdgeUnion( $N$ ,  $H$ )

Given networks  $N$  and  $H$  having the same number of vertices, construct their edge union  $K$ . This construction identifies the  $i$ -th vertex of  $N$  with the  $i$ -th vertex of  $H$  for all  $i$ . The edge union has the same vertex-set as  $N$  (and hence as  $H$ ) and there is an edge  $[u, v]$  with capacity  $c$  in  $K$  if and only if either there is an edge  $[u, v]$  with capacity  $c$  in  $N$  or if there is an edge  $[u, v]$  with capacity  $c$  in  $H$ . The resulting network has standard support and the inherited edge capacities but neither vertex labels, edge labels nor weights.

## 150.6 Conversion Functions

Conversion functions do not preserve a graph's support and vertex/edge decorations. That is, the resulting graph has standard support and no vertex/edge decorations. A slight exception to this rule occurs when the resulting graph is a network (of type `GrphNet`) and is described in detail in `UnderlyingNetwork`.

### 150.6.1 Orientated Graphs

The rules followed in building an orientated graph from an undirected graph are the same as those described for simple graphs (see Section 149.8).

#### **OrientatedGraph( $G$ )**

Given a multigraph  $G$ , produce a multidigraph  $D$  whose vertex-set is the same as that of  $G$  and whose edge-set consists of the edges of  $G$ , each given a direction. The edges of  $D$  are always directed from the lower numbered vertex to the higher numbered vertex. Thus, if  $G$  contains the edge  $\{u, v\}$ , then  $D$  will have the edge  $[u, v]$  if  $u < v$ , otherwise the edge  $[v, u]$ . If  $G$  has a loop at  $u$ , then  $D$  will have a directed loop at  $u$ .

### 150.6.2 Converse

#### **Converse( $G$ )**

Given a multidigraph  $G$  with edge-set  $E$ , produce a multidigraph  $D$  whose vertex-set is the same as that of  $G$  and whose edge-set is  $\{[u, v] : [v, u] \in E\}$ .

### 150.6.3 Converting between Simple Graphs and Multigraphs

Any simple (di)graph can be converted into a multi(di)graph and any multi(di)graph can be converted into a simple (di)graph. The resulting graph has standard support and neither vertex labels nor edge decorations, unless it is a network, in which case all the edges in the resulting graph are assigned a capacity of 1 (0 if loops).

Let  $G$  be a graph and  $e$  an edge of  $G$  from  $u$  to  $v$  and let  $H$  be the graph resulting from the conversion. If  $G$  and  $H$  are both undirected or both directed then  $e$  is also an edge of  $H$ . If  $G$  is undirected while  $H$  is undirected then both edges  $[u, v]$  and  $[v, u]$  are edges of  $H$ . If  $G$  is directed while  $H$  is directed then the edge  $\{u, v\}$  is an edge of  $H$ .

Since these conversion functions do not retain the original graph's support and vertex/edge decorations, they may also be used when requiring a copy of a graph  $G$  without  $G$ 's support and vertex/edge decorations.

#### **UnderlyingGraph( $G$ )**

The underlying simple graph of the graph  $G$ . The support and vertex/edge decorations of  $G$  are not retained.

#### **UnderlyingDigraph( $G$ )**

The underlying simple digraph of the graph  $G$ . The support and vertex/edge decorations of  $G$  are not retained.

#### **UnderlyingMultiGraph( $G$ )**

The underlying multigraph of the graph  $G$ . The support and vertex/edge decorations of  $G$  are not retained.

**UnderlyingMultiDigraph( $G$ )**

The underlying multidigraph of the graph  $G$ . The support and vertex/edge decorations of  $G$  are not retained.

**UnderlyingNetwork( $G$ )**

The underlying network of the graph  $G$ . The support and vertex/edge decorations of  $G$  are not retained except when  $G$  is a network in which case only the edge capacities are retained. If  $G$  is not a network, then all the edge capacities are set to 1 (0 for loops).

## 150.7 Elementary Invariants and Predicates for Multigraphs

Most but not all of the invariants and predicates that apply to simple graphs (see Sections 149.10 and 149.11) also apply to multigraphs. We list them below.

Let  $G$  and  $H$  be two graphs. For clarity, we list here once again the conditions under which  $G$  is equal to  $H$  and  $H$  is a subgraph of  $G$ .

The graphs  $G$  and  $H$  are equal if and only if:

- they are of the same type,
- they are structurally identical,
- they have the same support,
- they have identical vertex and edge labels,
- if applicable, the total capacity from  $u$  to  $v$  in  $G$  is equal to the total capacity from  $u$  to  $v$  in  $H$ .

Also,  $H$  is a subgraph of  $G$  if and only if:

- they are of the same type,
- $H$  is a structural subgraph of  $G$ ,
- any vertex  $v$  in  $H$  has the same support as the vertex  $\text{VertexSet}(G)!v$  in  $G$ ,
- any vertex  $v$  in  $H$  has the same label as the vertex  $\text{VertexSet}(G)!v$  in  $G$ ,
- any edge  $e$  in  $H$  has the same label as the edge  $\text{EdgeSet}(G)!e$  in  $G$ ,
- if applicable, the total capacity from  $u$  to  $v$  in  $G$  is at least as large as the total capacity from  $u$  to  $v$  in  $H$ .

Note that the truth value of the above two tests is not dependent on the weights of the edges of the graphs, should these edges be weighted.

Finally, we have introduced a few predicates to help users determine if a general graph is simple or not, undirected or not.



Order( $G$ )

NumberOfVertices( $G$ )

The number of vertices of the graph  $G$ .

Size( $G$ )

NumberOfEdges( $G$ )

The number of edges of the graph  $G$ .

$u \text{ adj } v$

Let  $u$  and  $v$  be two vertices of the same graph  $G$ . If  $G$  is undirected, returns **true** if and only if  $u$  and  $v$  are adjacent. If  $G$  is directed, returns **true** if and only if there is an edge directed from  $u$  to  $v$ .

$e \text{ adj } f$

Let  $e$  and  $f$  be two edges of the same graph  $G$ . If  $G$  is undirected, returns **true** if and only if  $e$  and  $f$  share a common vertex. If  $G$  is directed, returns **true** if and only if the terminal vertex of  $e$  ( $f$ ) is the initial vertex of  $f$  ( $e$ ).

$u \text{ notadj } v$

The negation of the **adj** predicate applied to vertices.

$e \text{ notadj } f$

The negation of the **adj** predicate applied to edges.

$u \text{ in } e$

Let  $u$  be a vertex and  $e$  an edge of a graph  $G$ . Returns **true** if and only if  $u$  is an end-vertex of  $e$ .

$u \text{ notin } e$

The negation of the **in** predicate applied to a vertex with respect to an edge.

$G \text{ eq } H$

Returns **true** if and only if the graphs  $G$  and  $H$  are equal, that is if and only if they are structurally equal and are compatible with respect to their support, vertex and edge labels, and edge capacities (see the introduction to this section).

IsSubgraph( $G$ ,  $H$ )

Returns **true** if and only if  $H$  is a subgraph of  $G$ , that is, if and only if  $H$  is a structural subgraph of  $G$  and the graphs are compatible with respect to their support, vertex and edge labels, and edge capacities (see the introduction to this section).

IsBipartite( $G$ )

Returns **true** if and only if the graph  $G$  is bipartite.

**Bipartition( $G$ )**

Given a bipartite graph  $G$ , return its two partite sets in the form of a pair of subsets of  $V(G)$ .

**IsRegular( $G$ )**

Returns **true** if and only if  $G$  is a regular graph.

**IsComplete( $G$ )**

Returns **true** if and only if the graph  $G$ , on  $n$  vertices, is the complete graph on  $n$  vertices.

**IsEmpty( $G$ )**

Returns **true** if and only if the edge-set of the graph is empty.

**IsNull( $G$ )**

Returns **true** if and only if the vertex-set of the graph is empty.

**IsSimple( $G$ )**

Returns **true** if and only if  $G$  is a simple graph.

**IsUndirected( $G$ )**

Returns **true** if and only if  $G$  is a undirected graph.

**IsDirected( $G$ )**

Returns **true** if and only if  $G$  is a directed graph.

## 150.8 Adjacency and Degree

The adjacency and degree functionalities that apply to simple graphs (see 149.12) similarly apply to multigraphs.

### 150.8.1 Adjacency and Degree Functions for Multigraphs

**Degree( $u$ )**

Given a vertex  $u$  of a graph  $G$ , return the degree of  $u$ , ie the number of edges incident to  $u$ .

**Alldeg( $G, n$ )**

Given a multigraph  $G$ , and a non-negative integer  $n$ , return the set of all vertices of  $G$  that have degree equal to  $n$ .

**MaximumDegree( $G$ )**

**Maxdeg( $G$ )**

The maximum of the degrees of the vertices of the multigraph  $G$ . This function returns two values: the maximum degree, and a vertex of  $G$  having that degree.

**MinimumDegree( $G$ )**

**Mindeg( $G$ )**

The minimum of the degrees of the vertices of the multigraph  $G$ . This function returns two values: the minimum degree, and a vertex of  $G$  having that degree.

**DegreeSequence( $G$ )**

Given a multigraph  $G$  such that the maximum degree of any vertex of  $G$  is  $r$ , return a sequence  $D$  of length  $r + 1$ , such that  $D[i]$ ,  $1 \leq i \leq r + 1$ , is the number of vertices in  $G$  having degree  $i - 1$ .

**Neighbours( $u$ )**

**Neighbors( $u$ )**

Given a vertex  $u$  of a graph  $G$ , return the set of vertices of  $G$  that are adjacent to  $u$ .

**IncidentEdges( $u$ )**

Given a vertex  $u$  of a graph  $G$ , return the set of all edges incident with the vertex  $u$ .

## 150.8.2 Adjacency and Degree Functions for Multidigraphs

**InDegree( $u$ )**

The number of edges directed into the vertex  $u$  belonging to a multidigraph.

**OutDegree( $u$ )**

The number of edges of the form  $[u, v]$  where  $u$  is a vertex belonging to a multidigraph.

**MaximumInDegree( $G$ )**

**Maxindeg( $G$ )**

The maximum indegree of the vertices of the multidigraph  $G$ . This function returns two values: the maximum indegree, and the first vertex of  $G$  having that degree.

**MinimumInDegree( $G$ )**

**Minindeg( $G$ )**

The minimum indegree of the vertices of the multidigraph  $G$ . This function returns two values: the minimum indegree, and the first vertex of  $G$  having that degree.

**MaximumOutDegree( $G$ )**

**Maxoutdeg( $G$ )**

The maximum outdegree of the vertices of the multidigraph  $G$ . This function returns two values: the maximum outdegree, and the first vertex of  $G$  having that degree.

**MinimumOutDegree( $G$ )**

**Minoutdeg( $G$ )**

The minimum outdegree of the vertices of the multidigraph  $G$ . This function returns two values: the minimum outdegree, and the first vertex of  $G$  having that degree.

**Degree( $u$ )**

Given a vertex  $u$  belonging to the multidigraph  $G$ , return the total degree of  $u$ , i.e. the sum of the in-degree and out-degree for  $u$ .

**MaximumDegree( $G$ )**

**Maxdeg( $G$ )**

The maximum total degree of the vertices of the multidigraph  $G$ . This function returns two values: the maximum total degree, and the first vertex of  $G$  having that degree.

MinimumDegree( $G$ )

Mindeg( $G$ )

The minimum total degree of the vertices of the multidigraph  $G$ . This function returns two values: the minimum total degree, and the first vertex of  $G$  having that degree.

Alldeg( $G$ ,  $n$ )

Given a multidigraph  $G$ , and a non-negative integer  $n$ , return the set of all vertices of  $G$  that have total degree equal to  $n$ .

DegreeSequence( $G$ )

Given a multidigraph  $G$  such that the maximum degree of any vertex of  $G$  is  $r$ , return a sequence  $D$  of length  $r + 1$ , such that  $D[i]$ ,  $1 \leq i \leq r + 1$ , is the number of vertices in  $G$  having degree  $i - 1$ .

InNeighbours( $u$ )

InNeighbors( $u$ )

Given a vertex  $u$  of a multidigraph  $G$ , return the set containing all vertices  $v$  such that  $[v, u]$  is an edge in  $G$ , i.e. the initial vertex of all edges that are directed into the vertex  $u$ .

OutNeighbours( $u$ )

OutNeighbors( $u$ )

Given a vertex  $u$  of the multidigraph  $G$ , return the set of vertices  $v$  of  $G$  such that  $[u, v]$  is an edge in  $G$ , i.e. the set of vertices  $v$  that are terminal vertices of edges directed from  $u$  to  $v$ .

IncidentEdges( $u$ )

Given a vertex  $u$  of a graph  $G$ , return the set of all edges incident with the vertex  $u$ , that is, the set of all edges incident into  $u$  and incident from  $u$ .

## 150.9 Connectedness

All the functions relating to connectivity issues are the same for simple graphs and multigraphs. See Section 149.13.1 and its subsections for specific details about the algorithms underlying these functions.

### 150.9.1 Connectedness in a Multigraph

**IsConnected( $G$ )**

Returns **true** if and only if the undirected graph  $G$  is a connected graph.

**Components( $G$ )**

The connected components of the undirected graph  $G$ . These are returned in the form of a sequence of subsets of the vertex-set of  $G$ .

**Component( $u$ )**

The subgraph corresponding to the connected component of the multigraph  $G$  containing vertex  $u$ . The support and vertex/edge decorations are *not* retained in the resulting (structural) subgraph.

**IsSeparable( $G$ )**

Returns **true** if and only if the graph  $G$  is connected and has at least one cut vertex.

**IsBiconnected( $G$ )**

Returns **true** if and only if the graph  $G$  is biconnected. The graph  $G$  must be undirected.

**CutVertices( $G$ )**

The set of cut vertices for the connected undirected graph  $G$  (as a set of vertices).

**Bicomponents( $G$ )**

The biconnected components of the undirected graph  $G$ . These are returned in the form of a sequence of subsets of the vertex-set of  $G$ . The graph may be disconnected.

### 150.9.2 Connectedness in a Multidigraph

**IsStronglyConnected( $G$ )**

Returns **true** if and only if the multidigraph  $G$  is strongly connected.

**IsWeaklyConnected( $G$ )**

Returns **true** if and only if the multidigraph  $G$  is weakly connected.

**StronglyConnectedComponents( $G$ )**

The strongly connected components of the multidigraph  $G$ . These are returned in the form of a sequence of subsets of the vertex-set of  $G$ .

**Component( $u$ )**

The subgraph corresponding to the connected component of the multidigraph  $G$  containing vertex  $u$ . The support and vertex/edge decorations are *not* retained in the resulting (structural) subgraph.

### 150.9.3 Triconnectivity for Multigraphs

We refer the reader to Subsection 149.13.3 of the graphs chapter for details about the triconnectivity algorithm implemented here, and especially about the meaning of the `Splitcomponents` function. The triconnectivity algorithm applies to undirected graphs only.

**IsTriconnected(*G*)**

Returns `true` if and only if  $G$  is triconnected. The graph  $G$  must be undirected.

**Splitcomponents(*G*)**

The split components of the undirected graph  $G$ . These are returned in the form of a sequence of subsets of the vertex-set of  $G$ . The graph may be disconnected. The second return value returns the cut vertices and the separation pairs as sequences of one or two vertices respectively.

**SeparationVertices(*G*)**

The cut vertices and/or the separation pairs of the undirected graph  $G$  as a sequence of sequences of one and/or two vertices respectively. The graph may be disconnected. The second return value returns the split components of  $G$ .

### 150.9.4 Maximum Matching in Bipartite Multigraphs

We refer the reader to Subsection 149.13.4 for information about the maximum matching algorithm.

**MaximumMatching(*G* : *parameters*)**

**Al**

**MONSTG**

*Default* : “*PushRelabel*”

A maximum matching in the bipartite graph  $G$ . The matching is returned as a sequence of edges of  $G$ . The parameter **Al** enables the user to select the algorithm which is to be used: **Al** := “*PushRelabel*” or **Al** := “*Dinic*”.

### 150.9.5 General Vertex and Edge Connectivity in Multigraphs and Multidigraphs

See Subsection 149.13.5 for details about the algorithms underlying the functions listed below. These functions apply to both undirected and directed graphs.

**VertexSeparator(*G* : *parameters*)**

**Al**

**MONSTG**

*Default* : “*PushRelabel*”

If  $G$  is an undirected graph, a *vertex separator* for  $G$  is a smallest set of vertices  $S$  such that for any  $u, v \in V(G)$ , every path connecting  $u$  and  $v$  passes through at least one vertex of  $S$ .

If  $G$  is a directed graph, a vertex separator for  $G$  is a smallest set of vertices  $S$  such that for any  $u, v \in V(G)$ , every directed path from  $u$  to  $v$  passes through at least one vertex of  $S$ .

**VertexSeparator** returns the vertex separator of  $G$  as a sequence of vertices of  $G$ . The parameter **A1** enables the user to select the algorithm which is to be used: **A1** := "PushRelabel" or **A1** := "Dinic".

<b>VertexConnectivity</b> ( $G$ : <i>parameters</i> )
---

<b>A1</b>	<b>MONSTG</b>	<i>Default</i> : "PushRelabel"
-----------	---------------	--------------------------------

Returns the vertex connectivity of the graph  $G$ , the size of a minimum vertex separator of  $G$ . Also returns a vertex separator for  $G$  as the second value. The parameter **A1** enables the user to select the algorithm which is to be used: **A1** := "PushRelabel" or **A1** := "Dinic".

<b>IsKVertexConnected</b> ( $G, k$ : <i>parameters</i> )
--

<b>A1</b>	<b>MONSTG</b>	<i>Default</i> : "PushRelabel"
-----------	---------------	--------------------------------

Returns **true** if the vertex connectivity of the graph  $G$  is at least  $k$ , **false** otherwise. The parameter **A1** enables the user to select the algorithm which is to be used: **A1** := "PushRelabel" or **A1** := "Dinic".

<b>EdgeSeparator</b> ( $G$ : <i>parameters</i> )
--

<b>A1</b>	<b>MONSTG</b>	<i>Default</i> : "PushRelabel"
-----------	---------------	--------------------------------

If  $G$  is an undirected graph, an *edge separator* for  $G$  is a smallest set of edges  $T$  such that for any  $u, v \in V(G)$ , every path connecting  $u$  and  $v$  passes through at least one edge of  $T$ .

If  $G$  is a directed graph, an edge separator for  $G$  is a smallest set of edges  $T$  such that for any  $u, v \in V(G)$ , every directed path from  $u$  to  $v$  passes through at least one edge of  $T$ .

**EdgeSeparator** returns the edge separator of  $G$  as a sequence of edges of  $G$ . The parameter **A1** enables the user to select the algorithm which is to be used: **A1** := "PushRelabel" or **A1** := "Dinic".

<b>EdgeConnectivity</b> ( $G$ : <i>parameters</i> )
---

<b>A1</b>	<b>MONSTG</b>	<i>Default</i> : "PushRelabel"
-----------	---------------	--------------------------------

Returns the edge connectivity of the graph  $G$ , the size of a minimum edge separator of  $G$ . Also returns as the second value an edge separator for  $G$ . The parameter **A1** enables the user to select the algorithm which is to be used: **A1** := "PushRelabel" or **A1** := "Dinic".

<b>IsKEdgeConnected</b> ( $G, k$ : <i>parameters</i> )
--

<b>A1</b>	<b>MONSTG</b>	<i>Default</i> : "PushRelabel"
-----------	---------------	--------------------------------

Returns **true** if the edge connectivity of the graph  $G$  is at least  $k$ , **false** otherwise. The parameter **A1** enables the user to select the algorithm which is to be used: **A1** := "PushRelabel" or **A1** := "Dinic".



**Example H150E9**

---

We demonstrate that the connectivity functions deal properly with multiple edges.

```
> G := MultiGraph< 3 | < 1, {2, 3} >, < 1, {2, 3} >, < 2, {2, 3} > >;
> G;
Multigraph
Vertex  Neighbours
1       3 2 3 2 ;
2       3 2 2 1 1 ;
3       2 1 1 ;
> EdgeConnectivity(G);
3
> EdgeSeparator(G);
[ < {3, 2}, 11 >, < {1, 2}, 5 >, < {1, 2}, 1 > ]
```

---

**150.10 Spanning Trees**

All the trees returned by the functions described below are returned as structural subgraphs of the original graph whose support and vertex and edge decorations are not retained in the resulting tree.

**SpanningTree(G)**

Given a connected undirected graph  $G$ , construct a spanning tree for  $G$  rooted at an arbitrary vertex of  $G$ . The spanning tree is returned as a structural subgraph of  $G$ , without the support, vertex or edge decorations of  $G$ .

**SpanningForest(G)**

Given a graph  $G$ , construct a spanning forest for  $G$ . The forest is returned as a structural subgraph of  $G$ , without the support, vertex or edge decorations of  $G$ .

**BreadthFirstSearchTree(u)****BFSTree(u)**

Given a vertex  $u$  belonging to the graph  $G$ , return a breadth-first search for  $G$  rooted at the vertex  $u$ . The tree is returned as a structural subgraph of  $G$ , without the support, vertex or edge decorations of  $G$ . Note that  $G$  may be disconnected.

DepthFirstSearchTree(*u*)

DFSTree(*u*)

Given a vertex  $u$  belonging to the graph  $G$ , return a depth-first search tree  $T$  for  $G$  rooted at the vertex  $u$ . The tree  $T$  is returned as a structural subgraph of  $G$ , without the support, vertex or edge decorations of  $G$ . Note that  $G$  may be disconnected.

The fourth return argument returns, for each vertex  $u$  of  $G$ , the tree order of  $u$ , that is, the order in which the vertex  $u$  has been visited while performing the depth-first search. If  $T$  does not span  $G$  then the vertices of  $G$  not in  $T$  are given tree order from  $\text{Order}(T) + 1$  to  $\text{Order}(G)$ .

## 150.11 Planar Graphs

The planarity algorithm implemented in MAGMA tests whether an undirected graph or multigraph is planar. If the graph is planar, then an embedding of the graph is produced, otherwise a Kuratowski subgraph is identified. For a thorough discussion of this algorithm, its implementation and complexity, the reader is referred to Section 149.21.

IsPlanar( $G$ )

Tests whether the (undirected) graph  $G$  is planar. The graph may be disconnected. If the graph is non-planar then a Kuratowski subgraph of  $G$  is returned: That is, a subgraph of  $G$  homeomorphic to  $K_5$  or  $K_{3,3}$ . The support and vertex/edge decorations of  $G$  are *not* retained in this (structural) subgraph.

Obstruction( $G$ )

Returns a Kuratowski obstruction if the graph is non-planar, or the empty graph if the graph is planar. The Kuratowski graph is returned as a (structural) subgraph of  $G$ ; the support and vertex/edge decorations are not retained.

IsHomeomorphic( $G$ : *parameters*)

**Graph**

MONSTG

*Default :*

Tests if a graph is homeomorphic to either  $K_5$  or  $K_{3,3}$ . The parameter **Graph** must be set to either “K5” or “K33”; it has no default setting.

Faces( $G$ )

Returns the faces of the planar graph  $G$  as sequences of the edges bordering the faces of  $G$ . If  $G$  is disconnected, then the face defined by an isolated vertex  $v$  is given as  $[v]$ .

**Face(u, v)**

Returns the face of the planar graph  $G$  bordered by the directed edge  $[u, v]$  as an ordered list of edges of  $G$ .

Note that a directed edge and an orientation determine a face uniquely: We can assume without loss of generality that the plane is given a clockwise orientation. Then given a directed edge  $e = [u_1, v_1]$ , the face defined by  $e$  is the ordered set of edges  $[u_1, v_1], [u_2, v_2], \dots, [u_m, v_m]$  such that  $v_i = u_{i+1}$  for all  $i$ ,  $1 \leq i < m$ ,  $v_m = u_1$ , and for each  $v_i = u_{i+1}$ , the neighbours of  $v_i$ ,  $u_i$  and  $v_{i+1}$ , are *consecutive* vertices in  $v_i$ 's adjacency list whose order is *anti-clockwise*.

**Face(e)**

Let  $e$  be the edge  $u, v$  of the planar graph  $G$  (recall that  $G$  is undirected). Then **Face**( $u, v$ ) returns the face bordered by the directed edge  $[u, v]$  as a sequence of edges of  $G$ .

**NFaces(G)****NumberOfFaces(G)**

Returns the number of faces of the planar graph  $G$ . In the case of a disconnected graph, an isolated vertex counts for one face.

**Embedding(G)**

Returns the planar embedding of the graph  $G$  as a sequence  $S$  where  $S[i]$  is a sequence of edges incident from vertex  $i$ .

**Embedding(v)**

Returns the ordered list of edges (in clockwise order say) incident from vertex  $v$ .

**Example H150E10**

The purpose of the example is to show the embedding and faces of a graph with multiples edges and loops.

```
> G := MultiGraph< 3 | < 1, {2, 3} >, < 1, {2} >, < 2, {2, 3} > >;
> G;
Multigraph
Vertex Neighbours
1      2 3 2 ;
2      3 2 2 1 1 ;
3      2 1 ;
> IsPlanar(G);
true
> Faces(G);
[
  [ < {1, 2}, 5 >, < {2, 1}, 1 > ],
  [ < {1, 2}, 1 >, < {2, 2}, 7 >, < {2, 3}, 9 >, < {3, 1}, 3 > ],
  [ < {1, 3}, 3 >, < {3, 2}, 9 >, < {2, 1}, 5 > ],
```

```

    [ < {2, 2}, 7 > ]
  ]
> Embedding(G);
[
  [ < {1, 2}, 5 >, < {1, 2}, 1 >, < {1, 3}, 3 > ],
  [ < {2, 3}, 9 >, < {2, 2}, 7 >, < {2, 1}, 1 >, < {2, 1}, 5 > ],
  [ < {3, 1}, 3 >, < {3, 2}, 9 > ]
]

```

---

**Example H150E11**

We show how to construct the dual graph  $D$  of a planar graph  $G$  and how to find all its minimal cuts. The vertex set of the dual is the set of faces  $F$  of  $G$  where face  $f_i$  is adjacent to face  $f_j$  if and only if  $f_i$  and  $f_j$  share a common edge in  $G$ . For the purpose of this example a cut of a graph  $G$  is defined as a set of edges which disconnects  $G$ .

Let us construct a small planar graph  $G$  and its dual  $D$ . For clarity, the support of  $D$  will be the standard support (we could have chosen it to be the set of faces of  $G$ ).

```

> G := MultiGraph< 4 | {1, 2}, {1, 2}, {1, 3}, {2, 3}, {2, 4}, {3, 4}>;
> IsPlanar(G);
true
> Faces(G);
[
  [ < {1, 3}, 5 >, < {3, 2}, 7 >, < {2, 1}, 3 > ],
  [ < {1, 2}, 3 >, < {2, 1}, 1 > ],
  [ < {1, 2}, 1 >, < {2, 4}, 9 >, < {4, 3}, 11 >, < {3, 1}, 5 > ],
  [ < {3, 4}, 11 >, < {4, 2}, 9 >, < {2, 3}, 7 > ]
]
> F := {@ SequenceToSet(f) : f in Faces(G) @} ;
> D := MultiGraph< #F | >;
> mapG2D := [ 0 : i in [1..Max(Indices(G))] ];
> mapD2G := [ 0 : i in [1..Max(Indices(G))] ];
> for u in VertexSet(D) do
>   for v in VertexSet(D) do
>     if Index(v) le Index(u) then
>       continue;
>     end if;
>     M := F[ Index(u) ] meet F[ Index(v) ];
>     for e in M do
>       D, edge :=
>         AddEdge(D, VertexSet(D)!u, VertexSet(D)!v);
>
>       mapG2D[Index(e)] := Index(edge);
>       mapD2G[Index(edge)] := Index(e);
>     end for;
>   end for;
> end for;
>

```

```

> e_star := map< EdgeSet(G) -> EdgeSet(D) |
> x :-> EdgeSet(D).mapG2D[Index(x)],
> y :-> EdgeSet(G).mapD2G[Index(y)] >;

```

The map `e_star` is the bijection from  $G$ 's edge-set into  $D$ 's edge-set:

```

> for e in EdgeSet(G) do
>   e, "    ", e @ e_star;
> end for;
< {1, 3}, 5 >      < {1, 3}, 3 >
< {1, 2}, 3 >      < {1, 2}, 1 >
< {1, 2}, 1 >      < {2, 3}, 7 >
< {2, 4}, 9 >      < {3, 4}, 11 >
< {2, 3}, 7 >      < {1, 4}, 5 >
< {3, 4}, 11 >     < {3, 4}, 9 >
>
> for e in EdgeSet(D) do
>   e, "    ", e @@ e_star;
> end for;
< {1, 4}, 5 >      < {2, 3}, 7 >
< {1, 3}, 3 >      < {1, 3}, 5 >
< {1, 2}, 1 >      < {1, 2}, 3 >
< {2, 3}, 7 >      < {1, 2}, 1 >
< {3, 4}, 11 >     < {2, 4}, 9 >
< {3, 4}, 9 >      < {3, 4}, 11 >

```

If  $G$  is biconnected, then any of its faces is bounded by a cycle. From Euler's formula giving the number of faces in a graph, we deduce that the boundaries of the internal faces of  $G$ , which form a chordless cycle, form a basis for the cycle space of  $G$ .

It is a well-known fact that, if  $G$  is connected and planar, a set of edges  $E$  is the set of edges of a cycle in  $G$  if and only if  $E^* = \{e^* : e \in E\}$  is a minimal cut in  $D$ . For more details, see [Die00, § 4].

From this we conclude that we can compute the minimal cuts generating the cut space of the dual graph  $D$ . We verify that  $G$  is biconnected, we compute the cut corresponding to each face of  $G$ , and verify that it is a minimal cut. All the cuts together form a generating set of the cut space of  $D$ . Had we not included the cut corresponding to the external face of  $G$ , we would have a basis of the cut space.

```

> IsBiconnected(G);
true
> for f in F do
>   Cut := { e @ e_star : e in f };
>   H := D;
>   RemoveEdges(~H, Cut);
>   assert not IsConnected(H);
>
>   for e in Cut do
>     C := Exclude(Cut, e);
>     H := D;
>     RemoveEdges(~H, C);

```

```

>      assert IsConnected(H);
>      end for;
> end for;

```

---

## 150.12 Distances, Shortest Paths and Minimum Weight Trees

Two standard algorithms have been implemented for finding single-source shortest paths in weighted graphs. One is Dijkstra's algorithm for graphs without negative weight cycles, the second is Bellman-Ford's for those graphs with negative weight cycles. Dijkstra's algorithm is implemented either by a priority queue (binary heap) or a Fibonacci heap. The Fibonacci heap is asymptotically faster for sparse graphs. But for most practical purposes (graphs of small order) the binary heap outperforms the Fibonacci heap and is therefore chosen as the default data structure wherever relevant.

Johnson's algorithm has been chosen for the all-pairs shortest paths computation. Indeed, it outperforms the simpler Floyd's algorithm, especially as the graphs get larger.

Finally, Prim's algorithm is used to implement the minimum weight tree computation for undirected graphs. (The tree is a spanning tree if and only if the graph is connected.)

All the functions described below apply to general graphs whose edges are assigned a weight. If the graph under consideration is not weighted, then all its edges are assumed to have weight one. To assign weights to the edges of a graph, see Subsection 150.4.2. Note that all the functions described below accept negatively weighted edges.

Reachable( <i>u</i> , <i>v</i> : <i>parameters</i> )
--

UseFibonacciHeap

BOOL

Default : false

Return true if and only there is a path from vertex *u* to vertex *v*. If true, also returns the distance between *u* and *v*.

Distance( <i>u</i> , <i>v</i> : <i>parameters</i> )
---

UseFibonacciHeap

BOOL

Default : false

Given vertices *u* and *v* in a graph *G*, computes the distance from *u* to *v*. Results in an error if there is no path in *G* from *u* to *v*.

Distances( <i>u</i> : <i>parameters</i> )
---

UseFibonacciHeap

BOOL

Default : false

Given a vertex *u* in a graph *G*, computes the sequence *D* of distances from *u* to *v*, *v* any vertex in *G*. Given any vertex *v* in *G*, let *i* be Index(*v*). Then *D*[*i*], if defined, is the distance from *u* to *v*. If there is no path from *u* to *v*, then the sequence element *D*[*i*] is undefined.

PathExists( $u, v : parameters$ )
-----------------------------------

UseFibonacciHeap	BOOL
------------------	------

Default : false

Return **true** if and only there is a path from vertex  $u$  to vertex  $v$  in the parent graph  $G$ . If so, also returns a shortest path from  $u$  to  $v$  as a sequence of edges of  $G$ .

Path( $u, v : parameters$ )
-----------------------------

ShortestPath( $u, v : parameters$ )
-------------------------------------

UseFibonacciHeap	BOOL
------------------	------

Default : false

Given vertices  $u$  and  $v$  in a graph  $G$ , computes a shortest path from  $u$  to  $v$  as a sequence of edges of  $G$ . Results in an error if there is no path in  $G$  from  $u$  to  $v$ .

Paths( $u : parameters$ )
---------------------------

ShortestPaths( $u : parameters$ )
-----------------------------------

UseFibonacciHeap	BOOL
------------------	------

Default : false

Given a vertex  $u$  in a graph  $G$ , computes the sequence  $P$  of shortest paths from  $u$  to  $v$ , for every vertex  $v$  in  $G$ . Given any vertex  $v$  in  $G$ , let  $i$  be  $\text{Index}(v)$ . If there exists a path from  $u$  to  $v$  then  $P[i]$  is a sequence of edges giving a shortest path from  $u$  to  $v$ . If there is no path from  $u$  to  $v$ , then the sequence element  $P[i]$  is undefined.

GeodesicExists( $u, v : parameters$ )
---------------------------------------

UseFibonacciHeap	BOOL
------------------	------

Default : false

Return **true** if and only there is a path from vertex  $u$  to vertex  $v$  in the parent graph  $G$ . If true, also returns a shortest path from  $u$  to  $v$  as a sequence of vertices of  $G$ .

Geodesic( $u, v : parameters$ )
---------------------------------

UseFibonacciHeap	BOOL
------------------	------

Default : false

Given vertices  $u$  and  $v$  in a graph  $G$ , this function computes a shortest path from  $u$  to  $v$  as a sequence of vertices of  $G$ . An error results if there is no path in  $G$  from  $u$  to  $v$ .

Geodesics( $u : parameters$ )
-------------------------------

UseFibonacciHeap	BOOL
------------------	------

Default : false

Given a vertex  $u$  in a graph  $G$ , this function computes the sequence  $P$  of shortest paths from  $u$  to  $v$ , for every vertex  $v$  in  $G$ . Given any vertex  $v$  in  $G$ , let  $i$  be  $\text{Index}(v)$ . If there exists a path from  $u$  to  $v$  then  $P[i]$  is a sequence of vertices specifying a shortest path from  $u$  to  $v$ . If there is no path from  $u$  to  $v$ , then the sequence element  $P[i]$  is undefined.

HasNegativeWeightCycle( $u : parameters$ )
--

Return **true** if and only if there is a negative-weight cycle reachable from vertex  $u$ .

HasNegativeWeightCycle(G)

Return **true** if and only if the graph  $G$  has negative-weight cycles.

AllPairsShortestPaths(G : parameters)

UseFibonacciHeap

BOOL

Default : false

Computes the all-pairs shortest paths. Let  $u$  and  $v$  be two vertices of a graph  $G$  and let  $i = \text{Index}(u)$  and  $j = \text{Index}(v)$ . Let  $S_1$  and  $S_2$  be the two sequences returned by **AllPairsShortestPaths**, and let  $s_1 = S_1[i]$  and  $s_2 = S_2[i]$ . Then  $s_1[j]$ , if defined, gives the distance from  $u$  to  $v$  and  $s_2[j]$ , if defined, gives the vertex preceding  $v$  in the shortest path from  $u$  to  $v$ . An error results if the graph  $G$  has a negative-weight cycle.

MinimumWeightTree(u : parameters)

UseFibonacciHeap

BOOL

Default : false

Returns a minimum weight tree rooted at vertex  $u$ ,  $u$  any vertex of an *undirected* graph  $G$ , as a subgraph of  $G$ . The tree spans  $G$  if and only if  $G$  is connected. The support of  $G$  as well as the vertex and edge decorations in  $G$  are transferred to the tree.

### Example H150E12

---

We create a weighted multidigraph.

```
> G := MultiDigraph< 5 | [1, 2], [1, 2], [1, 3], [2, 4], [3, 5], [3, 4], [4, 5] >;
> E := EdgeSet(G);
> AssignWeight(~G, E.1, 1);
> AssignWeight(~G, E.2, 5);
> AssignWeight(~G, E.3, 10);
> AssignWeight(~G, E.4, 1);
> AssignWeight(~G, E.5, -5);
> AssignWeight(~G, E.6, 1);
> AssignWeight(~G, E.7, 2);
>
>
> V := VertexSet(G);
> E := EdgeSet(G);
> for e in E do
>   e, " ", Weight(e);
> end for;
< [1, 3], 3 >      10
< [1, 2], 2 >      5
< [1, 2], 1 >      1
< [2, 4], 4 >      1
< [3, 4], 6 >      1
< [3, 5], 5 >     -5
```



```
< [4, 5], 7 >      2
```

We verify that it has no negative weight cycle reachable from vertex 1, and that there is a path from vertex 1 to vertex 5.

```
> HasNegativeWeightCycle(V!1);
false
> b, d := Reachable(V!1, V!5);
> assert b;
> P := Path(V!1, V!5);
> G := Geodesic(V!1, V!5);
>
> d;
4
> P;
[ < [1, 2], 1 >, < [2, 4], 4 >, < [4, 5], 7 > ]
> G;
[ 1, 2, 4, 5 ]
```

Finally, we verify that the shortest path found has length 4.

```
> dP := 0;
> for e in P do
>   dP += Weight(e);
> end for;
> assert dP eq d;
```

Note that had we taken instead an undirected graph, we would have had to assign only positive weights to the edges of the graph: any undirected edge  $\{u, v\}$  with negative weight results in the negative weight cycles  $\{u, u\}$  and  $\{v, v\}$ .

---

### Example H150E13

We create a weighted multigraph with one edge assigned a negative weight.

```
> G := MultiGraph< 5 | {1, 2}, {1, 2}, {1, 3}, {2, 4}, {3, 5}, {3, 4}, {4, 5} >;
> E := EdgeSet(G);
> AssignWeight(~G, E.1, 1);
> AssignWeight(~G, E.3, 5);
> AssignWeight(~G, E.5, 10);
> AssignWeight(~G, E.7, 1);
> AssignWeight(~G, E.9, -5);
> AssignWeight(~G, E.11, 1);
> AssignWeight(~G, E.13, 2);
>
> V := VertexSet(G);
> E := EdgeSet(G);
> for e in E do
>   e, " ", Weight(e);
> end for;
< {1, 3}, 5 >      10
```

```

< {1, 2}, 3 >    5
< {1, 2}, 1 >    1
< {2, 4}, 7 >    1
< {3, 4}, 11 >   1
< {3, 5}, 9 >   -5
< {4, 5}, 13 >   2

```

We compute a minimum weight spanning tree rooted at vertex 1.

```

> T := MinimumWeightTree(V!1);
> ET := EdgeSet(T);
> for e in ET do
>   e, " ", Weight(e);
> end for;
< {1, 2}, 1 >    1
< {2, 4}, 5 >    1
< {3, 5}, 7 >   -5
< {3, 4}, 3 >    1

```

We compute any other spanning tree rooted at vertex 1 (say a depth first tree using `DFSTree`), and verify that the weights of the edges in  $G$  corresponding to the edges of the tree sum up to a total weight which is no smaller than the weight of the minimum weight spanning tree.

```

> DFST := DFSTree(V!1);
> EDT := EdgeSet(DFST);
> for e in EDT do
>   u := InitialVertex(e);
>   v := TerminalVertex(e);
>   w := Min([ Weight(edge) : edge in Edges(V!u, V!v) ]);
>   e, " ", w;
> end for;
< {1, 3}, 1 >   10
< {2, 4}, 7 >    1
< {3, 4}, 3 >    1
< {4, 5}, 5 >    2

```

### 150.13 Bibliography

[Die00] Reinhard Diestel. *Graph Theory, Second Edition*. Springer, 2000.

# 151 NETWORKS

<b>151.1 Introduction . . . . .</b>	<b>5049</b>	<b>+=</b>	<b>5058</b>
<b>151.2 Construction of Networks .</b>	<b>5049</b>	AddEdge(N, u, v, c)	5058
Network< >	5050	AddEdge(N, u, v, c, l)	5058
Network< >	5050	AddEdges(N, S)	5058
151.2.1 Magma Output: Printing of a Net-		AddEdge(~N, u, v, c)	5058
work . . . . .	5051	AddEdge(~N, u, v, c, l)	5058
		AddEdges(~N, S)	5058
<b>151.3 Standard Construction for</b>		151.3.3 Union of Networks . . . . .	5058
<b>Networks . . . . .</b>	<b>5053</b>	<b>151.4 Maximum Flow and Minimum</b>	
151.3.1 Subgraphs . . . . .	5053	<b>Cut . . . . .</b>	<b>5059</b>
sub< >	5053	MinimumCut(s, t : -)	5061
151.3.2 Incremental Construction: Adding		MinimumCut(Ss, Ts : -)	5061
Edges . . . . .	5057	MaximumFlow(s, t : -)	5062
+	5057	MaximumFlow(Ss, Ts : -)	5062
+	5058	Flow(e)	5062
+	5058	Flow(u, v)	5062
+=	5058	<b>151.5 Bibliography . . . . .</b>	<b>5065</b>
+=	5058		



# Chapter 151

## NETWORKS

### 151.1 Introduction

Networks are an essential tool in modelling communication systems and dependence problems. A network is generally defined as a directed graph whose arcs are associated with a cost and a capacity; it may have multiple (i.e., parallel) edges.

The fundamental network flow problem is the minimum cost flow problem, that is, determining a maximum flow at minimum cost from a specified source to a specified sink. Specializations of this problem are the shortest path problem (where there is no capacity constraint) and the maximum flow problem (where there is no cost constraint). Some of the related problems are the minimum spanning tree problem (finding a spanning tree whose sum of the costs of its arcs is minimum), the matching problem (a pairing of the edges of the graph according to some criteria), and the multicommodity flow problem (where arcs may carry several flows of different nature). For a comprehensive monograph on networks, their implementation and applications, see [RAO93].

For convenience we provide users with the MAGMA network object with type `GrphNet`. It differs from the MAGMA multidigraph object having type `GrphMultDir` in one respect only: its edges are always assumed to be capacitated. The edges of a network have a capacity of one by default, unless they are specifically assigned a capacity. The loops in a network always have capacity zero. Since networks are a specialisation of multidigraphs, all the functions applying to multidigraphs also apply to networks. Below we outline those few functions that specifically concern networks, namely, their construction.

### 151.2 Construction of Networks

Networks are constructed in a similar way to multidigraphs (Subsection 150.2.2). In this implementation the order  $n$  of a network is bounded by 134217722. See Section 149.2.1 for more details on this.

Let  $N$  be the network to be constructed. In all cases, whenever an edge  $[u, v]$ ,  $u \neq v$ , is to be added to  $N$ , its capacity will be set to 1 (0 if a loop) unless either its capacity is explicitly given at construction time, or it is the edge of a network, in which case the capacity of the edge remains as it was in the original network.

As an example, if  $D$  is a digraph, then the edges of the network  $N$  constructed as `N := Network< Order(D) | D >`; will be all the edges of  $D$  whose capacity is set as 1 (or 0 if they are loops).

Network< n   edges >
Network< S   edges >

Construct the network  $N$  with vertex-set  $V = \{@v_1, v_2, \dots, v_n@\}$  (where  $v_i = i$  for each  $i$  if the first form of the constructor is used, or the  $i$ th element of the enumerated or indexed set  $S$  otherwise), and edge-set  $E = \{e_1, e_2, \dots, e_q\}$ . This function returns three values: The network  $N$ , the vertex-set  $V$  of  $N$ ; and the edge-set  $E$  of  $N$ .

The elements of  $E$  are specified by the list *edges*, where the items of *edges* may be objects of the following types:

- (a) A pair  $[v_i, v_j]$  of vertices in  $V$ . The directed edge  $[v_i, v_j]$  from  $v_i$  to  $v_j$  with capacity 1 (or 0 if it is a loop) will be added to the edge-set for  $N$ .
- (b) A tuple of the form  $\langle v_i, N_i \rangle$  where  $N_i$  will be interpreted as a set of out-neighbours for the vertex  $v_i$ . The elements of the sets  $N_i$  must be elements of  $V$ . If  $N_i = \{u_1, u_2, \dots, u_r\}$ , the edges  $[v_i, u_1], \dots, [v_i, u_r]$  will be added to  $N$ , all with capacity 1 (or 0 if they are loops).
- (c) A tuple of the form  $\langle [v_i, v_j], c \rangle$  where  $v_i, v_j$  are vertices in  $V$  and  $c$  the non-negative capacity of the directed edge  $[v_i, v_j]$  added to  $N$ .
- (d) A sequence  $[N_1, N_2, \dots, N_n]$  of  $n$  sets, where  $N_i$  will be interpreted as a set of out-neighbours for the vertex  $v_i$ . All the edges  $[v_i, u_i]$ ,  $u_i \in N_i$ , are added to  $N$  with capacity 1 (or 0 if they are loops).

In addition to these four basic ways of specifying the *edges* list, the items in *edges* may also be:

- (e) An edge  $e$  of a graph (or di/multi/multidigraph) or network of order  $n$ . If  $e$  is an edge of a network  $H$ , then it will be added to  $N$  with the capacity it has in  $H$ . If  $e$  is not a network edge, then it will be added to  $N$  with capacity 1, or 0 if it is a loop.
- (f) An edge-set  $E$  of a graph (or di/multi/multidigraph) or network of order  $n$ . Every edge  $e$  in  $E$  will be added to  $N$  according to the rule set out for a single edge.
- (g) A graph (or di/multi/multidigraph) or network  $H$  of order  $n$ . Every edge  $e$  in  $H$ 's edge-set is added to  $N$  according to the rule set out for a single edge.
- (h) A  $n \times n$  (0, 1)-matrix  $A$ . The matrix  $A$  will be interpreted as the adjacency matrix for a digraph  $H$  on  $n$  vertices and the edges of  $H$  will be included among the edges of  $N$  with capacity 1 (0 if loops).
- (i) A set of
  - (i) Pairs of the form  $[v_i, v_j]$  of vertices in  $V$ .
  - (ii) Tuples of the form  $\langle v_i, N_i \rangle$  where  $N_i$  will be interpreted as a set of out-neighbours for the vertex  $v_i$ .
  - (iii) A tuple of the form  $\langle [v_i, v_j], c \rangle$  where  $v_i, v_j$  are vertices in  $V$  and  $c$  a non-negative capacity.

- (iv) Edges of a graph (or di/multi/multidigraph) or network of order  $n$ .
- (v) Graphs (or di/multi/multidigraphs) or networks of order  $n$ .
- (j) A sequence of
  - (i) Tuples of the form  $\langle v_i, N_i \rangle$  where  $N_i$  will be interpreted as a set of out-neighbours for the vertex  $v_i$ .
  - (ii) A tuple of the form  $\langle [v_i, v_j], c \rangle$  where  $v_i, v_j$  are vertices in  $V$  and  $c$  a non-negative capacity.

---

**Example H151E1**

We construct a network from a digraph, and observe that the edges that are not loops have a capacity of 1:

```
> SetSeed(1, 0);
> n := 5;
> d := 0.2;
> D := RandomDigraph(n, d : SparseRep := true);
> N := Network< n | D >;
> D;
Digraph
Vertex Neighbours
1      2 1 ;
2      3 2 ;
3      ;
4      5 ;
5      2 ;
> N;
Network
Vertex Neighbours
1      1 [ 0 ] 2 [ 1 ] ;
2      2 [ 0 ] 3 [ 1 ] ;
3      ;
4      5 [ 1 ] ;
5      2 [ 1 ] ;
```

---

### 151.2.1 Magma Output: Printing of a Network

MAGMA displays a network  $N$  in the form of a list of vertices, each accompanied by a list of its outgoing capacitated edges (each followed by the capacity of the edge in brackets). Thus, in the previous example H151E1, it can be verified that all edges have capacity 1 (since the network was constructed from a digraph) except those edges that are loops.

If the network has multiple edges from  $u$  to  $v$ , then *each* edge from  $u$  to  $v$ , or rather its end-point  $v$ , is printed followed by the capacity of that edge. Also, the end-points in the adjacency list are not ordered and appear in the order in which they were created. The next example illustrates these two points.

**Example H151E2**

We construct a network from a set of tuples  $\langle [vertex, vertex], capacity \rangle$  and we exhibit a multiple edge.

```

> n := 5;
> C := 5;
> M := 3;
> T := [];
> for i in [1..12] do
>   u := Random(1, n);
>   v := Random(1, n);
>   m := Random(1, M);
>   for j in [1..m] do
>     c := Random(0, C);
>     if u eq v then
>       Append(~T, < [u, u], 0 >);
>     else
>       Append(~T, < [u, v], c >);
>     end if;
>   end for;
> end for;
> T;
[
  <[ 5, 4 ], 1>, <[ 5, 4 ], 3>, <[ 5, 4 ], 2>, <[ 5, 4 ], 1>,
  <[ 5, 4 ], 5>, <[ 1, 3 ], 2>, <[ 1, 3 ], 2>, <[ 5, 5 ], 0>,
  <[ 5, 5 ], 0>, <[ 2, 1 ], 2>, <[ 4, 2 ], 2>, <[ 4, 2 ], 5>,
  <[ 4, 2 ], 1>, <[ 4, 1 ], 3>, <[ 4, 1 ], 4>, <[ 4, 1 ], 3>,
  <[ 2, 3 ], 1>, <[ 2, 3 ], 3>, <[ 4, 3 ], 5>, <[ 4, 3 ], 3>,
  <[ 4, 3 ], 4>, <[ 2, 2 ], 0>, <[ 2, 2 ], 0>, <[ 5, 4 ], 0>,
  <[ 4, 4 ], 0>
]
> N := Network< n | T >;
> N;
Network
Vertex Neighbours
1      3 [ 2 ] 3 [ 2 ] ;
2      2 [ 0 ] 2 [ 0 ] 3 [ 3 ] 3 [ 1 ] 1 [ 2 ] ;
3      ;
4      4 [ 0 ] 3 [ 4 ] 3 [ 3 ] 3 [ 5 ] 1 [ 3 ] 1 [ 4 ] 1 [ 3 ] 2
      [ 1 ] 2 [ 5 ] 2 [ 2 ] ;
5      4 [ 0 ] 5 [ 0 ] 5 [ 0 ] 4 [ 5 ] 4 [ 1 ] 4 [ 2 ] 4 [ 3 ] 4
      [ 1 ] ;
> Edges(N);
{@ < [1, 3], 6 >, < [1, 3], 7 >, < [2, 1], 10 >, < [2, 2], 22 >,
< [2, 2], 23 >, < [2, 3], 17 >, < [2, 3], 18 >, < [4, 1], 14 >,
< [4, 1], 15 >, < [4, 1], 16 >, < [4, 2], 11 >, < [4, 2], 12 >,
< [4, 2], 13 >, < [4, 3], 19 >, < [4, 3], 20 >, < [4, 3], 21 >,
< [4, 4], 25 >, < [5, 4], 1 >, < [5, 4], 2 >, < [5, 4], 3 >,

```



$\langle [5, 4], 4 \rangle, \langle [5, 4], 5 \rangle, \langle [5, 4], 24 \rangle, \langle [5, 5], 8 \rangle,$   
 $\langle [5, 5], 9 \rangle @\}$

---

## 151.3 Standard Construction for Networks

### 151.3.1 Subgraphs

The construction of a sub-network is very similar to the construction of a sub-multidigraph (see Subsection 150.5.1). Additional flexibility is available for setting edge capacities in the subgraph.

There are two constraints when building a subgraph (see the introduction to Section 150.7). Let  $N$  be a network, and  $H$  a subgraph of  $N$ . Then, given any vertices  $u, v$  of  $H$ , the edge multiplicity from  $u$  to  $v$  is no greater in  $H$  than it is in  $N$ , and the total capacity from  $u$  to  $v$  in  $H$  is no greater than the total capacity from  $u$  to  $v$  in  $N$ . Failure to satisfy these constraints will result in a run-time error when constructing a sub-network.

Assume that we intend to add an edge from  $u$  to  $v$  in  $H$ . Assume also that the total capacity from  $u$  to  $v$  in  $N$  is  $C_N$ , that the total capacity from  $u$  to  $v$  in  $H$  is  $C_H$  before adding the edge from  $u$  to  $v$  in  $H$ , and that the total capacity from  $u$  to  $v$  in  $H$  is  $C'_H$  after adding the edge from  $u$  to  $v$  in  $H$ . In order to satisfy the “capacity constraint” it is enough that  $C'_H \leq C_N$ , i.e. that  $C_H + c \leq C_N$  where  $c$  is the capacity of the edge one wants to add in  $H$ .

There are two methods for adding an edge from  $u$  to  $v$ . Firstly, adding the edge  $[u, v]$  to  $H$  without specifying its capacity in  $H$  assumes that the edge  $[u, v]$  will be added with capacity  $C_N$ . This implies that  $C_H$  is zero, since we require that  $C_H + C_N \leq C_N$ . Secondly, adding the edge  $[u, v]$  to  $H$  when specifying its capacity as  $c$  assumes that the edge  $[u, v]$  will be added with capacity  $c$  such that  $C_H + c \leq C_N$ .

Note that the support set, vertex labels and edge labels and weights, if applicable, are transferred from the network to the sub-network.

sub< N   list >
-----------------

Construct the network  $H$  as a subgraph (sub-network) of  $N$ . The function returns three values: The network  $H$ , the vertex-set  $V$  of  $H$ ; and the edge-set  $E$  of  $H$ . If  $N$  has a support set and/or if  $N$  has vertex/edge labels, and/or edge weights then *all* these attributes are transferred to the subgraph  $H$ . Edge capacities are also transferred to  $H$  unless they are specifically set as explained below.

The elements of  $V$  and of  $E$  are specified by the list *list* whose items can be objects of the following types:

- (a) A vertex of  $N$ . The resulting subgraph will be the subgraph induced on the subset of  $\text{VertexSet}(N)$  defined by the vertices in *list*.
- (b) An edge of  $N$ . The resulting subgraph will be the subgraph with vertex-set  $\text{VertexSet}(N)$  whose edge-set consists of the edges in *list* subject to the multiplicity and capacity constraints being satisfied.

- (c) A pair of  $[v_i, v_j]$  of vertices of  $N$ . The resulting subgraph will be the subgraph with vertex-set  $\text{VertexSet}(N)$  whose edge-set consists of the edges  $[v_i, v_j]$  in  $list$  whose capacity is assumed to be the total capacity from  $v_i$  to  $v_j$  in  $N$ . The multiplicity and capacity constraints must be satisfied.
- (d) A tuple of the form  $\langle [v_i, v_j], c \rangle$  where  $v_i, v_j$  are vertices of  $N$  and  $c$  the non-negative capacity of the edge  $[v_i, v_j]$  to be added to  $H$ . The resulting subgraph will be the subgraph with vertex-set  $\text{VertexSet}(N)$  whose edge-set consists of the edges as they are given in  $list$ , subject to the multiplicity and capacity constraints being satisfied.
- (e) A set of
  - (i) Vertices of  $N$ .
  - (ii) Edges of  $N$ .
  - (iii) Pairs of vertices of  $N$ .
  - (iv) Tuples of the form  $\langle [v_i, v_j], c \rangle$  where  $v_i, v_j$  are vertices of  $N$  and  $c$  the non-negative capacity of the edge  $[v_i, v_j]$  to be added to  $H$ .

---

**Example H151E3**

We start by constructing a network with some multiple edges.

```
> N := Network< 4 |
> < [1, 2], 2 >, < [1, 2], 3 >, < [1, 4], 5 >,
> < [2, 3], 1 >, < [2, 3], 3 >, < [3, 4], 1 >, < [3, 4], 6 > >;
> N;
Network
Vertex Neighbours
1      4 [ 5 ] 2 [ 3 ] 2 [ 2 ] ;
2      3 [ 3 ] 3 [ 1 ] ;
3      4 [ 6 ] 4 [ 1 ] ;
4      ;
> V := VertexSet(N);
> E := EdgeSet(N);
```

We construct a subgraph  $H$  of  $N$  induced by some of  $N$ 's vertices and we obtain the mapping from the vertices of  $N$  to the vertices of  $H$  and vice-versa.

```
> H := sub< N | V!1, V!3, V!4 >;
> assert IsSubgraph(N, H);
> H;
Network
Vertex Neighbours
1      3 [ 5 ] ;
2      3 [ 1 ] 3 [ 6 ] ;
3      ;
> V!VertexSet(H)!1, VertexSet(H)!V!1;
1 1
> V!VertexSet(H)!2, VertexSet(H)!V!3;
```

```

3 2
> V!VertexSet(H)!3, VertexSet(H)!V!4;
4 3

```

The next statements illustrate the “capacity constraint”: That is, given any pair  $[u, v]$  of vertices of  $H$ ,  $H$  a subgraph of  $N$ , the total capacity from  $u$  to  $v$  in  $H$  can not be greater than the total capacity from  $u$  to  $v$  in  $N$ . The subgraph constructor will fail whenever this rule cannot be satisfied. We give a few examples below.

```

> Edges(N);
{@ < [1, 2], 1 >, < [1, 2], 2 >, < [1, 4], 3 >, < [2, 3], 4 >, < [2, 3], 5 >,
< [3, 4], 6 >, < [3, 4], 7 > @}
> E.1, E.2;
< [1, 2], 1 > < [1, 2], 2 >
> Capacity(E.1);
2
> Capacity(E.2);
3
> Capacity(V!1, V!2);
5
>
> H := sub< N | E.1, E.1 >;
> H;
Network
Vertex  Neighbours
1       2 [ 2 ] 2 [ 2 ] ;
2       ;
3       ;
4       ;

```

Adding twice the edge  $E.1$  to  $H$  is a valid operation since the resulting total capacity from 1 to 2 in  $H$  is 4 while it is 5 in  $N$ .

```

> > H := sub< N | E.2, E.2 >;
>> H := sub< N | E.2, E.2 >;
    ^

```

```

Runtime error in sub< ... >: RHS argument 2 - Edge multiplicity and capacity
not compatible with subgraph constructor
>

```

Adding twice the edge  $E.2$  (which has capacity 3) to  $H$  would have resulted in the total capacity from 1 to 2 in  $H$  to be 6, while it is 5 in  $N$ .

```

> H := sub< N | E!< [1, 2], 1 >, E!< [1, 2], 1 > >;
> H;
Network
Vertex  Neighbours
1       2 [ 2 ] 2 [ 2 ] ;
2       ;
3       ;

```

```
4      ;
```

This succeeded since the total capacity from 1 to 2 in  $H$  is now 4.

```
> > H := sub< N | E!< [1, 2], 2 >, E!< [1, 2], 2 > >;
>> H := sub< N | E!< [1, 2], 2 >, E!< [1, 2], 2 > >;
      ^
```

```
Runtime error in sub< ... >: RHS argument 2 - Edge multiplicity and capacity
not compatible with subgraph constructor
```

```
>
```

Again, this operation failed as it would have resulted in the total capacity from 1 to 2 in  $H$  to be 6.

```
> H := sub< N | < [ V!1, V!2 ], 2 >, < [ V!1, V!2 ], 2 > >;
> H;
```

```
Network
```

```
Vertex Neighbours
1      2 [ 2 ] 2 [ 2 ] ;
2      ;
3      ;
4      ;
```

This operation is valid since the total capacity from 1 to 2 in  $H$  is now 4.

```
> > H := sub< N | < [ V!1, V!2 ], 2 >, < [ V!1, V!2 ], 4 > >;
>> H := sub< N | < [ V!1, V!2 ], 2 >, < [ V!1, V!2 ], 4 > >;
      ^
```

```
Runtime error in sub< ... >: RHS argument 2 - Tuple must be <[vertex,
vertex], capacity> with total edge multiplicity and capacity compatible with
subgraph constructor
```

```
>
```

This operation cannot succeed as the total capacity from 1 to 2 in  $H$  would be 6.

```
> H := sub< N | [ V!1, V!2 ] >;
> H;
```

```
Network
```

```
Vertex Neighbours
1      2 [ 5 ] ;
2      ;
3      ;
4      ;
```

Adding the edge  $[1,2]$  without specifying its capacity implies that an edge from 1 to 2 is added with capacity 5 to  $H$ , which is the total capacity from 1 to 2 in  $N$ .

```
> > H := sub< N | [ V!1, V!2 ], [ V!1, V!2 ] >;
>> H := sub< N | [ V!1, V!2 ], [ V!1, V!2 ] >;
      ^
```

```
Runtime error in sub< ... >: RHS argument 2 - Sequence must be
[vertex, vertex] with vertices of the LHS and must be unique
```

&gt;

Adding twice the edge  $[1, 2]$  without specifying its capacity would have resulted in the total capacity from 1 to 2 in  $H$  to be 10. This operation cannot succeed. Finally, let us illustrate the edge multiplicity constraint.

```
> > H := sub< N | E.4, E.4, E.4 >;
```

```
>> H := sub< N | E.4, E.4, E.4 >;
```

```
Runtime error in sub< ... >: RHS argument 3 - Edge multiplicity and capacity
not compatible with subgraph constructor
```

&gt;

Although the above statement satisfies the capacity constraint

```
> Capacity(E.4);
```

1

```
> Capacity(V!InitialVertex(E.4), V!TerminalVertex(E.4));
```

4

it cannot succeed since the edge multiplicity constraint is violated.

```
> EdgeMultiplicity(V!InitialVertex(E.4), V!TerminalVertex(E.4));
```

2

### 151.3.2 Incremental Construction: Adding Edges

Almost all the functions to add or remove either vertices or edges that are available for multidigraphs also apply to networks; they are not listed here, see Section 150.5.2 for details.

The only exception is the function `AddEdge(G, u, v, 1)` which differs for multidigraphs and networks. It is replaced by the functions `AddEdge(G, u, v, c)` and `AddEdge(G, u, v, c, 1)` which are specialised functions for adding capacitated edges to networks. There are a few more such specialised functions which are listed below, they all concern adding edges to a network.

Note that whenever an edge is added to a network using the general multidigraph functions, which do not allow specifying an edge capacity, the edge to be added is *always taken to have capacity 1 (or 0 if a loop)*.

$$N + \langle [u, v], c \rangle$$

Given two vertices  $u$  and  $v$  of a network  $N$ , and  $c$  a non-negative integer, adds an edge from  $u$  to  $v$  with capacity  $c$ . The support and edge capacities are retained. This function returns two values: The modified network, and the edge newly created (added). This feature is especially useful when adding parallel edges.

$$N + \{ \langle [u, v], c \rangle \}$$

$$N + [ \langle [u, v], c \rangle ]$$

Given tuples of the form  $\langle [u, v], c \rangle$ ,  $u$  and  $v$  vertices of the network  $N$  and  $c$  a non-negative integer, adds the edges from  $u$  to  $v$  with capacity  $c$ . Tuples can be contained in a set or a sequence; the latter is useful when dealing with duplicates. The support and edge capacities are retained.

$$N += \langle [u, v], c \rangle$$

$$N += \{ \langle [u, v], c \rangle \}$$

$$N += [ \langle [u, v], c \rangle ]$$

The procedural versions of the previous three functions. Tuples can be contained in a set or a sequence; the latter is useful when dealing with duplicates.

$$\text{AddEdge}(N, u, v, c)$$

Given two vertices  $u$  and  $v$  of the network  $N$ , and  $c$  a non-negative integer, adds an edge from  $u$  to  $v$  with capacity  $c$ . The support and edge capacities are retained. This function returns the modified network and the newly created edge. This feature is especially useful when adding parallel edges.

$$\text{AddEdge}(N, u, v, c, l)$$

Given two vertices  $u$  and  $v$  of the network  $N$ ,  $c$  a non-negative integer, and a label  $l$ , adds an edge from  $u$  to  $v$  with capacity  $c$  and label  $l$ . The support and edge capacities are retained. This function returns the modified network and the newly created edge. This feature is especially useful when adding parallel edges.

$$\text{AddEdges}(N, S)$$

Given a network  $N$  and a set or a sequence  $S$  of tuples, this function includes the edges specified in  $S$ . The tuples must be of the form  $\langle [u, v], c \rangle$ , where  $u$  and  $v$  vertices of  $N$  and  $c$  a non-negative integer. The support and existing vertex and edge decorations are retained.

$$\text{AddEdge}(\sim N, u, v, c)$$

$$\text{AddEdge}(\sim N, u, v, c, l)$$

$$\text{AddEdges}(\sim N, S)$$

Procedural versions of previous functions adding edges to a network. Tuples can be contained in a set or a sequence; the latter is useful when dealing with duplicates.

### 151.3.3 Union of Networks

It is possible to construct a new network from the union of two networks. For more details, we refer the reader to Subsection 150.5.4.

## 151.4 Maximum Flow and Minimum Cut

All the functions described in this section apply to general graphs whose edges are given a capacity, that is, networks. If the graph under consideration is not capacitated, then all its edges are assumed to have capacity one. To assign capacities to the edges of a graph, see Subsection 150.4.2. To create a MAGMA network object (with type `GrphNet`) see Section 151.2.

The fundamental network flow problem is the minimum cost flow problem, that is, determining a maximum flow at minimum cost from a specified source to a specified sink. Specializations of this problem are the shortest path problem (where there is no capacity constraint) which is covered in Section 150.12, and the maximum flow problem (where there is no cost constraint). Some of the related problems are the minimum spanning tree problem, the matching problem (a pairing of the edges of the graph according to some criteria), and the multicommodity flow problem (where arcs may carry several flows of different nature). For a comprehensive monograph on network problems, their implementation and applications, see [RAO93].

Let  $G$  be a general graph or multigraph whose edges are capacitated. If  $G$  is undirected then consider any edge  $\{u, v\}$  with capacity  $c$  as being equivalent to two directed edges  $[u, v]$  and  $[v, u]$ , both with capacity  $c$ . We may thus assume without loss of generality that  $G$  is directed, and from now on  $G$  is called a network. Let  $V$  and  $E$  be  $G$ 's vertex-set and edge-set respectively, and for every edge  $[u, v]$  in  $G$ , denote its capacity by  $c(u, v)$ . If there is no edge  $[u, v]$  in  $E$  we assume that  $c(u, v) = 0$ . By convention the capacity of an edge is a non-negative integer.

Distinguish two vertices of  $G$ , the *source*  $s$  and the *sink*  $t$ . A *flow* in  $G$  is an integer-valued function  $f : V \times V \rightarrow \mathbf{Z}$  that satisfies the following properties:

- (i) Capacity constraint:  $\forall u, v \in V, f(u, v) \leq c(u, v)$ .
- (ii) Skew symmetry:  $\forall u, v \in V, f(u, v) = -f(v, u)$ .
- (iii) Flow conservation:  $\forall u \in V \setminus \{s, t\}, \sum_{v \in V} f(u, v) = 0$ .

Note that the flow could have been defined as a real-valued function if the edge capacity had been defined as real-valued.

The quantity  $f(u, v)$ , which can be positive or negative, is called the *net flow* from vertex  $u$  to vertex  $v$ . The *value* of a flow  $f$  is defined as  $F = \sum_{v \in V} f(s, v)$ , that is, the total net flow out of the source.

In the *maximum flow problem* we wish to find a flow of maximum value from  $s$  to  $t$ . A *cut* of the network  $G$  with source  $s$  and sink  $t$  is a partition of  $V$  into  $S$  and  $T$  such that  $s \in S$  and  $t \in T$ . The *capacity* of the cut  $c(S)$  determined by  $S$  is the sum of the capacity  $c(u, v)$  of the edges  $[u, v]$  such that  $u \in S$  and  $v \in T$ . It is easy to see that  $F \leq c(S)$ , that is, the value of the flow from  $s$  into  $t$  cannot be larger than the capacity of any cut defined by  $s$  and  $t$ . In fact,  $F = c(S)$  if and only if  $F$  is a maximum flow and  $S$  is a cut of minimum capacity.

We have implemented two algorithms for finding the maximum flow in a network  $G$  from a source  $s$  to a sink  $t$ . One is the Dinic algorithm, the other is an instance of the

generic push-relabel algorithm. We have already encountered both algorithms in Subsections 149.13.4 and 149.13.5.

### The Dinic algorithm

The Dinic algorithm consists of two phases. In the first phase, one constructs a layered network which consists of all the “useful” edges of  $G$ : A useful edge  $[u, v]$  has the property that  $f(u, v) < c(u, v)$ . The first and the last layer of this layered network always contain  $s$  and  $t$  respectively. If such a layered network cannot be constructed then the flow in  $G$  is maximum and the algorithm ends.

In the second phase of the Dinic algorithm, one finds a maximal flow by constructing paths from  $s$  to  $t$  (using a depth-first search) in the layered network. Note that a maximal flow may not be maximum: A maximal flow satisfies the condition that for every path  $P$  from  $s$  to  $t$  in the layered network, there is at least a *saturated* edge in  $P$ . A saturated edge is one for which its flow equals its capacity. For more details on the Dinic algorithm, see [Eve79].

The Dinic algorithm has a theoretical complexity of  $O(|V|^2|E|)$ , which can be improved to  $O(|E|^{3/2})$  if  $G$  is a zero-one network (the capacities of the edges are either 0 or 1), or  $O(|V|^{2/3}|E|)$  if  $G$  is a zero-one network with no parallel edges. Prior to the advent of the push-relabel method (which we describe below), the Dinic algorithm was shown to be superior in practice to other methods. The MAGMA implementation of the Dinic algorithm only outperforms the push-relabel method for very sparse networks (a small number of edges relative to the number of vertices) whose edges have a small capacity and where the maximum flow is small. In other words, the Dinic method performs best on zero-one and very sparse networks.

### The push-relabel method

The generic push-relabel algorithm does not construct a flow by constructing paths from  $s$  to  $v$ . Rather, it starts by pushing the maximum possible flow out from the source  $s$  into the neighbours of  $s$ , then pushing the excess flow at those vertices into their own neighbours. This is repeated until all vertices of  $G$  except  $s$  and  $t$  have an excess flow of zero (that is, the flow conservation property is satisfied). Of course this might mean that some flow is pushed back into  $s$ .

At initialisation all vertices are given a height of 0 except  $s$  which will have height  $|V|$ , and flow is pushed out of  $s$ . The maximum flow that can be pushed out of  $s$  is  $\sum_{v \in V} c(s, v)$ . The idea is to push flow into vertices only in a “downward” manner, that is, from a vertex with height  $h + 1$  into a vertex with height  $h$ . If a vertex in  $V \setminus \{s, t\}$  has height  $h$ , it will be relabelled if it has an excess flow and none of its neighbours to which some flow could be pushed has height  $h - 1$ . Its new label (height) will be  $l + 1$  where  $l$  is the height of the lowest labelled neighbour towards which flow can be pushed. Flow may only be pushed along an edge  $[u, v]$  while the capacity constraint  $f(u, v) \leq c(u, v)$  is satisfied.

So the general notion is of pushing flow “downward” and “discharging” the vertices in  $V \setminus \{s, t\}$  from all non-zero excess flow they may have accumulated, a process which may require “lifting” some vertices so that discharging can proceed. It is easy to show that the



height of any vertex in  $V \setminus \{s, t\}$  is bounded by  $2|V| - 1$ , so the algorithm must terminate. When this happens, the flow is at a maximum.

The theoretical complexity of the generic push-relabel algorithm is  $O(|V|^2|E|)$  which can be improved to at least  $O(|V||E| \log(|V|^2/|E|))$  thanks to some heuristics.

One commonly used heuristic is the *global relabelling* procedure whereby the vertices in  $V \setminus \{s, t\}$  are pre-labelled according to their distance from the sink. Global relabelling may occur at specified intervals during execution. Another very useful heuristic is *gap relabelling* which involves recognising those vertices that have become unreachable from the sink and labelling them accordingly so that they won't be chosen during later execution. Finally, another heuristic involves the order in which the vertices to be discharged are processed.

When  $G$  is a one-zero network, choosing the vertex with smallest height works best, while choosing the vertex with largest height is recommended for general networks. These heuristics are fully described and analysed by B.V. Cherkassky and A.V. Goldberg *et al.* in [CGM<sup>+</sup>98] and [CG97]. The MAGMA implementation incorporates all the above heuristics together with some new ones.

As a rule, whenever the push-relabel algorithm is applied to a zero-one network (such as occurs when computing the connectivity of a graph in 149.13.5 or finding a maximum matching in a bipartite network in 149.13.4) the vertex with smallest height is chosen as the next vertex to be processed. When the algorithm is used to compute a maximum flow in a network (as below) the vertex with largest height is chosen.

For the two functions `MinimumCut` and `MaximumFlow` described below, the `PushRelabel` algorithm is used. It is only in the case of a very sparse zero-one network that `Dinic` may outperform `PushRelabel`.

<code>MinimumCut(s, t : parameters)</code>
--

A1

MONSTGEALT

Default : "PushRelabel"

Given vertices  $s$  and  $t$  of a network  $G$ , this function returns the subset  $S$  defining a minimum cut  $\{S, T\}$  of  $V$ , where  $s \in S$ ,  $t \in T$ , corresponding to the maximum flow  $F$  from  $s$  to  $t$ . The subset  $S$  is returned as a sequence of vertices of  $G$ . The maximum flow  $F$  is returned as a second value. The parameter A1 enables the user to select the algorithm to be used: A1 := "PushRelabel" or A1 := "Dinic".

<code>MinimumCut(Ss, Ts : parameters)</code>
--

A1

MONSTGEALT

Default : "PushRelabel"

Given two sequences  $S_s$  and  $T_s$  of vertices of a network  $G$ , this function returns the subset  $S$  defining a minimum cut  $\{S, T\}$  of  $V$ , such that  $S_s \subseteq S$  and  $T_s \subseteq T$ , and which corresponds to the maximum flow  $F$  from the vertices in  $S_s$  to the vertices in  $T_s$ . The subset  $S$  is returned as a sequence of vertices of  $G$ . The maximum flow  $F$  is returned as a second value. The parameter A1 enables the user to select the algorithm to be used: A1 := "PushRelabel" or A1 := "Dinic".

MaximumFlow( <i>s</i> , <i>t</i> : parameters)
--

Al

MONSTGELT

Default : “PushRelabel”

Given vertices  $s$  and  $t$  of a network  $G$ , this function returns the maximum flow  $F$  from  $s$  to  $t$ . The subset  $S$  defining a minimum cut  $\{S, T\}$  of  $\text{VertexSet}(N)$ ,  $s \in S$ ,  $t \in T$ , corresponding to  $F$  is returned as the second value. The subset  $S$  is returned as a sequence of vertices of  $G$ . The parameter **Al** enables the user to select the algorithm to be used: **Al** := “PushRelabel” or **Al** := “Dinic”.

MaximumFlow( <i>Ss</i> , <i>Ts</i> : parameters)
--

Al

MONSTGELT

Default : “PushRelabel”

Given two sequences  $S_s$  and  $T_s$  of vertices of a network  $G$ , this function returns the maximum flow  $F$  from the vertices in  $S_s$  to the vertices in  $T_s$ . The subset  $S$  defining a minimum cut  $\{S, T\}$  of  $\text{VertexSet}(N)$ ,  $S_s \subseteq S$  and  $T_s \subseteq T$ , corresponding to  $F$  is returned as the second value. The subset  $S$  is returned as a sequence of vertices of  $G$ . The parameter **Al** enables the user to select the algorithm to be used: **Al** := “PushRelabel” or **Al** := “Dinic”.

Flow( <i>e</i> )
------------------

Given an edge  $e$  in a network  $G$ , this function returns the flow of the edge  $e$  as an integer. In this instance we require that the edges of  $G$  be explicitly assigned capacities (see Subsection 150.4.2 or Section 151.2). Edges of  $G$  will carry a flow only if a flow has been constructed from a source to a sink in  $G$ . If no such flow has yet been constructed, all edges will have zero flow.

Flow( <i>u</i> , <i>v</i> )
-----------------------------

Let  $u$  and  $v$  be adjacent vertices of a network  $G$  whose edges have been explicitly assigned capacities (see Subsection 150.4.2 or Section 151.2). **Flow**(*u*, *v*) returns the total net flow from  $u$  to  $v$  as an integer. The total net flow from  $u$  to  $v$  is defined as the total outgoing flow from  $u$  into  $v$  minus the total ingoing flow into  $u$  from  $v$ . Thus the flow satisfies the skew symmetry **Flow**(*u*, *v*) = - **Flow**(*v*, *u*).

Edges of  $G$  will carry a flow only if a flow has been constructed from a source to a sink in  $G$ . If no such flow has yet been constructed, all edges will have zero flow.

### Example H151E4

---

We illustrate the maximum flow algorithm by applying it to find a maximum matching in a bipartite graph. This example exactly replicates the implementation of the maximum matching algorithm (Section 149.13.4).

We construct a bipartite graph.

```
> G := Graph< 7 | [ {5, 7}, {6, 7}, {4, 5, 6},
> {3}, {1, 3}, {2, 3}, {1, 2} ] : SparseRep := true >;
> assert IsBipartite(G);
> P := Bipartition(G);
> P;
[
```

```

    { 1, 2, 3 },
    { 4, 5, 6, 7 }
]

```

We add two extra vertices to  $G$ , the source  $s$  and the sink  $t$  and we construct three sets of pairs of vertices. The first contains all the pairs  $[s, u]$  where  $u \in P[1]$ , the second contains all the pairs  $[u, t]$  where  $u \in P[2]$ , and the third contains all the pairs  $[u, v]$  where  $u \in P[1]$  and  $v \in P[2]$ . From these sets we construct a multidigraph with capacitated edges.

```

> G += 2;
> G;
Graph
Vertex  Neighbours
1       7 5 ;
2       7 6 ;
3       6 5 4 ;
4       3 ;
5       3 1 ;
6       3 2 ;
7       2 1 ;
8       ;
9       ;
> s := Order(G)-1;
> t := Order(G);
>
> E1 := { [s, Index(u)] : u in P[1] };
> E2 := { [Index(u), t] : u in P[2] };
> E3 := { [Index(u), Index(v)] : u in P[1], v in P[2] };
>
> N := MultiDigraph< Order(G) | E1, E2, E3 >;
> N;
Multidigraph
Vertex  Neighbours
1       7 6 5 4 ;
2       5 4 7 6 ;
3       7 6 5 4 ;
4       9 ;
5       9 ;
6       9 ;
7       9 ;
8       1 3 2 ;
9       ;
> E := EdgeSet(N);
> for e in E do
>   AssignCapacity(~N, e, 1);

```

```
> end for;
```

First note that all the edges of  $N$  have been assigned capacity 1. Also, the edges of  $N$  are those connecting  $s$  to all the vertices in  $P[1]$ , those connecting all the vertices of  $P[2]$  to  $t$ , and those connecting all the vertices of  $P[1]$  to all the vertices of  $P[2]$ .

We construct a maximum flow  $F$  from  $s$  to  $t$  and we check that the capacity of the cut is indeed  $F$ .

```
> V := VertexSet(N);
> F := MaximumFlow(V!s, V!t);
> S := MinimumCut(V!s, V!t);
> F;
3
> S;
[ 8 ]
>
> c := 0;
> for u in S do
>   for v in OutNeighbours(u) do
>     if not v in S then
>       c += Capacity(u, v);
>       assert Capacity(u, v) eq Flow(u, v);
>     end if;
>   end for;
> end for;
> assert c eq F;
```

We now exhibit a matching in  $G$ . It will be given by the edges in  $E_3$  which are saturated, that is, whose flow is 1.

```
> M := [];
> for e in E3 do
>   u := V!e[1];
>   v := V!e[2];
>   if Flow(u, v) eq 1 then
>     E := Edges(u, v);
>     assert #E eq 1;
>     Append(~M, EndVertices(E[1]));
>   end if;
> end for;
> assert #M eq F;
> M;
[
  [ 1, 7 ],
  [ 3, 5 ],
  [ 2, 4 ]
```

```
]

```

Note that the statement

```
>      assert #E eq 1;

```

makes sense as the network  $N$  has no parallel edges.

We end the example by showing that the capacity constraint and the skew symmetry is satisfied by all edges in the network.

```
> for e in EdgeSet(N) do
>   u := EndVertices(e)[1];
>   v := EndVertices(e)[2];
>
>   assert Flow(u, v) le Capacity(u, v);
>   assert Flow(u, v) eq -Flow(v, u);
> end for;

```

Also, flow conservation holds for all vertices in  $V \setminus \{s, t\}$ .

```
> s := V!s;
> t := V!t;
> for u in V do
>   if not u eq s and not u eq t then
>     f := 0;
>     for v in OutNeighbours(u) do
>       E := Edges(u, v);
>       f += Flow(E[1]);
>     end for;
>     for v in InNeighbours(u) do
>       E := Edges(v, u);
>       f -= Flow(E[1]);
>     end for;
>
>     assert f eq 0;
>   end if;
> end for;

```

## 151.5 Bibliography

- [CG97] B.V. Cherkassky and A.V. Golberg. On Implementing the Push-Relabel Method for the Maximum Flow Problem. *Algorithmica*, 19:390–410, 1997.
- [CGM<sup>+</sup>98] B.V. Cherkassky, A.V. Golberg, Paul Martin, J.C. Setubal, and J. Stolfi. Augment or Push? A Computational Study of Bipartite Matching and Unit Capacity Flow Algorithms. Technical report, NEC Research Institute, 1998.
- [Eve79] Shimon Even. *Graph Algorithms*. Computer Science Press, 1979.
- [RAO93] T.L. Magnanti R.K. Ahuja and J.B. Orlin. *Network Flows, Theory, Algorithms, and Applications*. Prentice Hall, 1993.