RAMP

The Research Assistant for Maniplexes and Polytopes

0.5

21 July 2021

Gabe Cunningham

Mark Mixer

Gordon Williams

Gabe Cunningham
Email: gabriel.cunningham@umb.edu
Homepage: http://www.gabrielcunningham.com
Address: Gabe Cunningham
Department of Mathematics
University of Massachusetts Boston
100 William T. Morrissey Blvd.
Boston MA 02125

Mark Mixer
Email: mixerm@wit.edu

Gordon Williams
Email: giwilliams@alaska.edu
Homepage: http://williams.alaska.edu
Address: Gordon Williams
PO Box 756660
Department of Mathematics and Statistics
University of Alaska Fairbanks
Fairbanks, AK 99775-6660

Copyright

Acknowledgements

We appreciate very much all past and future comments, suggestions and contributions to this package and its documentation provided by GAP users and developers.

Contents

Chapter 1

Groups for Maps, Polytopes, and Maniplexes

1.1 Groups of Maps, Polytopes, and Maniplexes

1.1.1 AutomorphismGroup (for IsManiplex)

▷ `AutomorphismGroup(`*M*`)` (attribute)

Returns the automorphism group of *M*. This group is not guaranteed to be in any particular form. For particular permutation representations you should consider the various AutomorphismGroupOn... functions.

1.1.2 AutomorphismGroupFpGroup (for IsManiplex)

▷ `AutomorphismGroupFpGroup(`*M*`)` (attribute)

Returns the automorphism group of *M* as a finitely presented group.

1.1.3 AutomorphismGroupPermGroup (for IsManiplex)

▷ `AutomorphismGroupPermGroup(`*M*`)` (attribute)

Returns the automorphism group of *M* as a permutation group.

1.1.4 AutomorphismGroupOnFlags (for IsManiplex)

▷ `AutomorphismGroupOnFlags(`*M*`)` (attribute)

Returns the automorphism group of *M* as a permutation group action on the flags of *M*.

1.1.5 ConnectionGroup (for IsManiplex)

▷ `ConnectionGroup(`*M*`)` (attribute)

Returns the connection group of *M* as a permutation group. We may eventually allow other types of connection groups. Synonym: MonodromyGroup

1.1.6 EvenConnectionGroup (for IsManiplex)

▷ EvenConnectionGroup(*M*) (attribute)

Returns the even-word subgroup of the connection group of *M* as a permutation group.

1.1.7 RotationGroup (for IsManiplex)

▷ RotationGroup(*M*) (attribute)

Returns the rotation group of *M*. This group is not guaranteed to be in any particular form.

1.1.8 ChiralityGroup (for IsRotaryManiplex)

▷ ChiralityGroup(*M*) (attribute)

Returns the chirality group of the rotary maniplex *M*. This is the kernel of the group epimorphism from the rotation group of *M* to the rotation group of its maximal reflexible quotient. In particular, the chirality group is trivial if and only if *M* is reflexible.

1.1.9 ExtraRelators (for IsReflexibleManiplex)

▷ ExtraRelators(*M*) (attribute)

For a reflexible maniplex *M*, returns the relators needed to define its automorphism group as a quotient of the string Coxeter group given by its Schlafli symbol. Not particularly robust at the moment.

1.1.10 ExtraRotRelators (for IsRotaryManiplex)

▷ ExtraRotRelators(*M*) (attribute)

For a reflexible maniplex *M*, returns the relators needed to define its rotation group as a quotient of the rotation group of a string Coxeter group given by its Schlafli symbol. Not particularly robust at the moment.

1.1.11 IsManiplexable (for IsPermGroup)

▷ IsManiplexable(*permgroup*) (operation)

Returns: Boolean.

Given a permutation group, it asks if the generators could be the connection group of a maniplex. That is to say, are each of the generators and their products fixed point free.

1.2 Sggis

1.2.1 UniversalSggi

▷ `UniversalSggi(n)` (operation)
▷ `UniversalSggi(sym)` (operation)

Returns: `IsFpGroup`

In the first form, returns the universal Coxeter Group of rank n. In the second form, returns the Coxeter Group with Schlafli symbol sym.

1.2.2 Sggi (for IsList, IsList)

▷ `Sggi(symbol[, relations])` (operation)

Returns: `IsFpGroup`

Returns the sggi defined by the given Schlafli symbol and with the given relations. The relations can be given by a list of Tietze words or as a string of relators or relations that involve r0 etc. If no relations are given, then returns the universal sggi with the given Schlafli symbol.

Example

```
gap> g := Sggi([4,3,4], "(r0 r1 r2)^3, (r1 r2 r3)^3");;
gap> h := Sggi([4,4], "r0 = r2");;
gap> k := Sggi([infinity, infinity], [[1,2,1,2,1,2], [2,3,2,3,2,3]]);;
gap> k = Sggi([3,3]);
true
```

1.2.3 IsGgi (for IsGroup)

▷ `IsGgi(g)` (property)

Returns: whether g is generated by involutions. Or more specifically, whether GeneratorsOfGroup(g) all have order 2 or less.

1.2.4 IsStringy (for IsGroup)

▷ `IsStringy(g)` (property)

Returns: whether every pair of non-adjacent generators of g commute.

1.2.5 IsSggi (for IsGroup)

▷ `IsSggi(g)` (property)

Returns: whether g is a string group generated by involutions. Equivalent to `IsGgi(g)` and `IsStringy(g)`.

1.2.6 IsStringC (for IsGroup)

▷ `IsStringC(G)` (operation)

For an sggi `G`, returns whether the group is a string C group.

1.2.7 IsStringCPlus (for IsGroup)

▷ `IsStringCPlus(G)` (operation)

For a "string rotation group" *G*, returns whether the group is a string C+ group. It does not check whether *G* is a string rotation group.

1.2.8 SggiElement (for IsGroup, IsString)

▷ `SggiElement(g, str)` (operation)

Returns: the element of *g* with underlying word *str*.

Example

```
gap> g := Group((1,2),(2,3),(3,4));;
gap> SggiElement(g, "r0 r1");
(1,3,2)
```

For convenience, you can also use a reflexible maniplex M in place of g, in which case `AutomorphismGroup(M)` is used for g.

1.2.9 SggiFamily (for IsGroup, IsList)

▷ `SggiFamily(parent, words)` (operation)

Given a *parent* group and a list of strings that represent words in r0, r1, etc, returns a function. That function accepts a list of positive integers L, and returns the quotient of *parent* by the relations that set the order of each *words[i]* to L[i].

Example

```
gap> f := SggiFamily(Sggi([4,4]), ["r0 r1 r2 r1"]);
function( orders ) ... end
gap> g := f([3]);
<fp group on the generators [ r0, r1, r2 ]>
gap> Size(g);
72
gap> h := f([6]);
<fp group on the generators [ r0, r1, r2 ]>
gap> IsQuotient(h,g);
true
```

One of the advantages of building an SggiFamily is that testing whether one member of the family is a quotient of another member can be done quite quickly.

Chapter 2

Families of Polytopes

2.1 Classical polytopes

2.1.1 Vertex

▷ `Vertex()` (operation)

Returns: IsPolytope

Returns the universal 0-polytope.

Example

```
gap> Vertex();
UniversalPolytope(0)
```

2.1.2 Edge

▷ `Edge()` (operation)

Returns: IsPolytope

Returns the universal 1-polytope.

Example

```
gap> Edge();
UniversalPolytope(1)
```

2.1.3 Pgon (for IsInt)

▷ `Pgon(`*p*`)` (operation)

Returns: IsPolytope

Returns the p-gon.

Example

```
gap> Facets(Pgon(5));
[ UniversalPolytope(1) ]
```

2.1.4 Cube (for IsInt)

▷ `Cube(`*n*`)` (operation)

Returns: IsPolytope

Returns the n-cube.

Example

```
gap> Fvector(Cube(4));
[ 16, 32, 24, 8 ]
```

2.1.5 HemiCube (for IsInt)

▷ HemiCube(*n*) (operation)

Returns: IsPolytope

Returns the n-hemi-cube.

Example

```
gap> Fvector(HemiCube(4));
[ 8, 16, 12, 4 ]
```

2.1.6 CrossPolytope (for IsInt)

▷ CrossPolytope(*n*) (operation)

Returns: IsPolytope

Returns the n-cross-polytope.

Example

```
gap> NumberOfVertices(CrossPolytope(5));
10
```

2.1.7 HemiCrossPolytope (for IsInt)

▷ HemiCrossPolytope(*n*) (operation)

Returns: IsPolytope

Returns the n-hemi-cross-polytope.

Example

```
gap> NumberOfVertices(HemiCrossPolytope(5));
5
```

2.1.8 Simplex (for IsInt)

▷ Simplex(*n*) (operation)

Returns: IsPolytope

Returns the n-simplex.

Example

```
gap> Petrial(Simplex(3))=HemiCube(3);
true
```

2.1.9 CubicTiling (for IsInt)

▷ CubicTiling(*n*) (operation)

Returns: IsPolytope

Returns the rank $n+1$ polytope; the tiling of E^n by n-cubes.

Example

```
gap> SchlafliSymbol(CubicTiling(3));
[ 4, 3, 4 ]
```

2.1.10 Dodecahedron

▷ Dodecahedron() (operation)

Returns: IsPolytope

Returns the dodecahedron, {5, 3}.

Example

```
gap> Dual(Dodecahedron());
Icosahedron()
```

2.1.11 HemiDodecahedron

▷ HemiDodecahedron() (operation)

Returns: IsPolytope

Returns the hemi-dodecahedron, {5, 3}_5.

Example

```
gap> Dual(HemiDodecahedron());
ReflexibleManiplex([ 3, 5 ], "(r2*r1*r0)^5")
```

2.1.12 Icosahedron

▷ Icosahedron() (operation)

Returns: IsPolytope

Returns the icosahedron, {3, 5}.

Example

```
gap> Dual(Icosahedron());
Dodecahedron()
```

2.1.13 HemiIcosahedron

▷ HemiIcosahedron() (operation)

Returns: IsPolytope

Returns the hemi-icosahedron, {3, 5}_5.

Example

```
gap> Fvector(HemiIcosahedron());
[ 6, 15, 10 ]
```

2.1.14 24Cell

▷ 24Cell() (operation)

Returns: IsPolytope

Returns the 24-cell, {3, 4, 3}.

Example

```
gap> SchlafliSymbol(24Cell());
[ 3, 4, 3 ]
```

2.1.15 Hemi24Cell

▷ `Hemi24Cell()` (operation)

Returns: IsPolytope

Returns the hemi-24-cell, `{3, 4, 3}_6`.

Example

```
gap> SchlafliSymbol(Hemi24Cell());
[ 3, 4, 3 ]
```

2.1.16 120Cell

▷ `120Cell()` (operation)

Returns: IsPolytope

Returns the 120-cell, {5, 3, 3}.

Example

```
gap> NumberOfIFaces(120Cell(),3);
120
```

2.1.17 Hemi120Cell

▷ `Hemi120Cell()` (operation)

Returns: IsPolytope

Returns the hemi-120-cell, `{5, 3, 3}_15`.

Example

```
gap> NumberOfIFaces(Hemi120Cell(),3);
60
```

2.1.18 600Cell

▷ `600Cell()` (operation)

Returns: IsPolytope

Returns the 600-cell, `{3, 3, 5}`.

Example

```
gap> Dual(600Cell());
120Cell()
```

2.1.19 Hemi600Cell

▷ `Hemi600Cell()` (operation)

Returns: IsPolytope

Returns the hemi-600-cell, `{3, 3, 5}_15`.

Example

```
gap> Dual(Hemi600Cell())=Hemi120Cell();
true
```

2.1.20 BrucknerSphere

▷ `BrucknerSphere()` (operation)

Returns: IsPoset

Returns Bruckner's sphere.

Example

```
gap> IsLattice(BrucknerSphere());
true
```

2.1.21 InternallySelfDualPolyhedron1 (for IsInt)

▷ `InternallySelfDualPolyhedron1(p)` (operation)

Returns: IsPolytope

Constructs the internally self-dual polyhedron of type {p, p} described in Theorem 5.3 of [CM17]. #(https://doi.org/10.11575/cdm.v12i2.62785). p must be at least 7.

Example

```
gap> SchlafliSymbol(InternallySelfDualPolyhedron1(40));
[ 40, 40 ]
```

2.1.22 InternallySelfDualPolyhedron2 (for IsInt, IsInt)

▷ `InternallySelfDualPolyhedron2(p, k)` (operation)

Returns: IsPolytope

Constructs the internally self-dual polyhedron of type {p, p} described in Theorem 5.8 of [CM17].# (https://doi.org/10.11575/cdm.v12i2.62785). p must be even and at least 6, and k must be odd.

Example

```
gap> SchlafliSymbol(InternallySelfDualPolyhedron2(40,7));
[ 40, 40 ]
```

2.2 Flat and tight polytopes

2.2.1 FlatOrientablyRegularPolyhedron (for IsInt, IsInt, IsInt, IsInt)

▷ `FlatOrientablyRegularPolyhedron(p, q, i, j)` (operation)

Returns: polyhedron

polyhedron is the flat orientably regular polyhedron with automorphism group [p, q] / (r2 r1 r0 r1 = (r0 r1)^i (r1 r2)^j). This function validates the inputs to make sure that the polyhedron is well-defined. Use FlatOrientablyRegularPolyhedronNC if you do not want this validation.

2.2.2 FlatOrientablyRegularPolyhedraOfType (for IsList)

▷ `FlatOrientablyRegularPolyhedraOfType(sym)` (operation)

Returns a list of all flat, orientably regular polyhedra with Schlafli symbol *sym*.

2.2.3 TightOrientablyRegularPolytopesOfType (for IsList)

▷ `TightOrientablyRegularPolytopesOfType(sym)` (operation)

Returns a list of all tight, orientably regular polytopes with Schlafli symbol `sym`. When `sym` has length 2, this just calls FlatOrientablyRegularPolyhedraOfType(`sym`).

2.3 The Tomotope

2.3.1 Tomotope

▷ `Tomotope()` (operation)

Returns: maniplex

Constructs the *Tomotope* from [MPW12]

2.4 Toroids

2.4.1 ToroidalMap44

▷ `ToroidalMap44(u[, v])` (function)

Returns: IsManiplex

Returns the toroidal map $\{4,4\}_{\vec{u},\vec{v}}$. If only `u` is given, then `v` is taken to be `u` rotated 90 degrees, in which case the resulting map is either reflexible or chiral.

Example

```
gap> ToroidalMap44([3,0]) = ARP([4,4], "(r0 r1 r2 r1)^3");
true
gap> M := ToroidalMap44([1,2]);; IsChiral(M);
true
gap> ToroidalMap44([5,0]) = SmallestReflexibleCover(M);
true
gap> M := ToroidalMap44([2,0],[0,3]);; NumberOfFlagOrbits(M);
2
gap> M = ARP([4,4]) / "(r0 r1 r2 r1)^2, (r1 r0 r1 r2)^3";
true
gap> SmallestReflexibleCover(M) = ToroidalMap44([6,0]);
true
gap> ToroidalMap44([2,3],[4,1]) = ToroidalMap44([-3,2],[-1,4]);
true
```

2.4.2 CubicToroid (for IsInt,IsInt,IsInt)

▷ `CubicToroid(s, k, n)` (operation)

Returns: IsManiplex

Given IsInt triple `s, k, n`, will return the regular toroid $\{4,3^{n-2},4\}_{\vec{s}}$ where $\vec{s} = (s^k, 0^{n-k})$.

Example

```
gap> m44:=CubicToroid(3,2,2);;
gap> m44=ToroidalMap44([3,3]);
true
```

2.4.3 CubicToroid (for IsInt,IsList)

▷ `CubicToroid(n, vecs)` (operation)

Returns: IsManiplex

Given an integer n and a list of vectors *vecs*, returns the cubic toroid that is a quotient of CubicTiling(n) by the translation subgroup generated by the given vectors. The results may be nonsensical if *vecs* does not generate an n-dimensional translation group.

2.4.4 3343Toroid (for IsInt,IsInt)

▷ `3343Toroid(s, k)` (operation)

Returns: IsManiplex

Given IsInt pair *s*, *k*, will return the regular toroid $\{3,3,4,3\}_{\vec{s}}$ where $\vec{s} = (s^k, 0^{n-k})$. Note that k must be 0 or 1.

2.4.5 24CellToroid (for IsInt,IsInt)

▷ `24CellToroid(s, k)` (operation)

Returns: IsManiplex

Given IsInt pair *s*, *k*, will return the regular toroid $\{3,4,3,3\}_{\vec{s}}$ where $\vec{s} = (s^k, 0^{n-k})$. Note that k must be 0 or 1.

2.5 Uniform polyhedra

Representations of the uniform polyhedra here are from [HW10].

2.5.1 Cuboctahedron

▷ `Cuboctahedron()` (operation)

Returns: maniplex

Constructs the cuboctahedron.

Example

```
gap> SchlafliSymbol(Cuboctahedron());
[ [ 3, 4 ], 4 ]
```

2.5.2 TruncatedTetrahedron

▷ `TruncatedTetrahedron()` (operation)

Returns: maniplex

Constructs the truncated tetrahedron.

Example

```
gap> SchlafliSymbol(TruncatedTetrahedron());
[ [ 3, 6 ], 3 ]
```

2.5.3 TruncatedOctahedron

▷ TruncatedOctahedron() (operation)

Returns: maniplex

Constructs the truncated octahedron.

Example

```
gap> Fvector(TruncatedOctahedron());
[ 24, 36, 14 ]
```

2.5.4 TruncatedCube

▷ TruncatedCube() (operation)

Returns: maniplex

Constructs the truncated octahedron.

Example

```
gap> Fvector(TruncatedCube());
[ 24, 36, 14 ]
gap> SchlafliSymbol(TruncatedCube());
[ [ 3, 8 ], 3 ]
```

2.5.5 Icosadodecahedron

▷ Icosadodecahedron() (operation)

Returns: maniplex

Constructs the icosadodecahedron.

Example

```
gap> VertexFigure(Icosadodecahedron());
Pgon(4)
gap> Facets(Icosadodecahedron());
[ Pgon(5), Pgon(3) ]
```

2.5.6 TruncatedIcosahedron

▷ TruncatedIcosahedron() (operation)

Returns: maniplex

Constructs the truncated icosahedron.

Example

```
gap> Facets(TruncatedIcosahedron());
[ Pgon(6), Pgon(5) ]
```

2.5.7 SmallRhombicuboctahedron

▷ SmallRhombicuboctahedron() (operation)

Returns: maniplex

Constructs the small rhombicuboctahedron.

Example

```
gap> ZigzagVector(SmallRhombicuboctahedron());
[ 12, 8 ]
```

2.5.8 Pseudorhombicuboctahedron

▷ Pseudorhombicuboctahedron() (operation)

Returns: maniplex

Constructs the pseudorhombicuboctahedron.

Example

```
gap> Size(ConnectionGroup(Pseudorhombicuboctahedron()));
16072626615091200
```

2.5.9 SnubCube

▷ SnubCube() (operation)

Returns: maniplex

Constructs the snub cube.

Example

```
gap> IsEquivelar(PetrieDual(SnubCube()));
true
gap> SchlafliSymbol(PetrieDual(SnubCube()));
[ 30, 5 ]
gap> Size(ConnectionGroup(PetrieDual(SnubCube())));
3804202857922560
gap> Size(AutomorphismGroup(PetrieDual(SnubCube())));
24
```

2.5.10 SmallRhombicosidodecahedron

▷ SmallRhombicosidodecahedron() (operation)

Returns: maniplex

Constructs the small rhombicosidodecahedron.

Example

```
gap> Facets(SmallRhombicosidodecahedron());
[ Pgon(5), Pgon(4), Pgon(3) ]
```

2.5.11 GreatRhombicosidodecahedron

▷ GreatRhombicosidodecahedron() (operation)

Returns: maniplex

Constructs the great rhombicosidodecahedron.

Example

```
gap> Facets(GreatRhombicosidodecahedron());
[ Pgon(10), Pgon(4), Pgon(6) ]
```

2.5.12 SnubDodecahedron

▷ SnubDodecahedron() (operation)

Returns: maniplex

Constructs the small snub dodecahedron.

Example

```
gap> Facets(SnubDodecahedron());
[ Pgon(5), Pgon(3) ]
gap> IsEquivelar(PetrieDual(SnubDodecahedron()));
true
```

2.5.13 TruncatedDodecahedron

▷ `TruncatedDodecahedron()` (operation)

Returns: maniplex

Constructs the truncated dodecahedron.

2.5.14 GreatRhombicuboctahedron

▷ `GreatRhombicuboctahedron()` (operation)

Returns: maniplex

Constructs the great rhombicuboctahedron.

Chapter 3

Maniplexes

3.1 Constructors

3.1.1 ReflexibleManiplex

▷ `ReflexibleManiplex(`*g*`)` (operation)
▷ `ReflexibleManiplex(`*sym[, relations]*`)` (operation)

Returns: `IsReflexibleManiplex`

In the first form, we are given an Sggi *g* and we return the reflexible maniplex with that automorphism group, where the privileged generators are those returned by GeneratorsOfGroup(g).

Example

```
gap> g := Group([(1,2), (2,3), (3,4)]);
gap> M := ReflexibleManiplex(g);
gap> M = Simplex(3);
true
```

This function first checks whether g is an Sggi. Use `ReflexibleManiplexNC` to bypass that check. The second form returns the universal reflexible maniplex with Schlafli symbol *sym*. If the optional argument *relations* is given, then we return the reflexible maniplex with the given defining relations. The relations can be given by a list of Tietze words or as a string of relators or relations that involve r0 etc.

Example

```
gap> q := ReflexibleManiplex([4,3,4], "(r0 r1 r2)^3, (r1 r2 r3)^3");;
gap> q = ReflexibleManiplex([4,3,4], "(r0 r1 r2)^3 = (r1 r2 r3)^3 = 1");
true
gap> p := ReflexibleManiplex([infinity], "r0 r1 r0 = r1 r0 r1");;
```

If the option set_schlafli is set, then we set the Schlafli symbol to the one given. This may not be the correct Schlafli symbol, since the relations may cause a collapse, so this should only be used if you know that the Schlafli symbol is correct.

3.1.2 Maniplex (for IsPermGroup)

▷ `Maniplex(`*G*`)` (operation)

Returns: `IsManiplex`

Given a permutation group *G* on the set [1..N], returns a maniplex with N flags with connection group *G*. The output may not make sense if *G* is not an sggi.

Example
```
gap> G := Group([(1,2)(3,4)(5,6), (2,3)(4,5)(1,6)]);;
gap> M := Maniplex(G);
Pgon(3)
gap> c := ConnectionGroup(Cube(3));
<permutation group with 3 generators>
gap> Maniplex(c) = Cube(3);
true
```

3.1.3 Maniplex (for IsReflexibleManiplex, IsGroup)

▷ `Maniplex(M, H)` (operation)

Returns: `IsManiplex`

Let *M* be a reflexible maniplex and let *H* be a subgroup of AutomorphismGroup(*M*). This returns the maniplex *M/H*. This will be reflexible if and only if *H* is normal. For most purposes, it is probably easier to use QuotientManiplex, which takes a string of relations as input instead of a subgroup. The example below builds the map $\{4,4\}_{(1,0),(0,2)}$.

Example
```
gap> M := ReflexibleManiplex([4,4]);
CubicTiling(2)
gap> G := AutomorphismGroup(M);
<fp group of size infinity on the generators [ r0, r1, r2 ]>
gap> H := Subgroup(G, [G.1*G.2*G.3*G.2, (G.2*G.1*G.2*G.3)^2]);
Group([ r0*r1*r2*r1, (r1*r0*r1*r2)^2 ])
gap> M2 := Maniplex(M, H);
3-maniplex
gap> Size(M2);
16
```

3.1.4 Maniplex (for IsFunction, IsList)

▷ `Maniplex(F, inputs)` (operation)

Returns: `IsManiplex`

Constructs a formal maniplex, represented by an operation *F* and a list of arguments *inputs*. By itself, this does not really _do_ anything – it creates a maniplex object that only knows the operation *F* and the *inputs*. However, many polytope operations (such as Pyramid(M), Medial(M), etc) use this construction as a base, and then add "attribute computers" that tell the formal maniplex how to compute certain things in terms of properties of the base. See AddAttrComputer for more information.

3.1.5 Maniplex (for IsPoset)

▷ `Maniplex(P)` (operation)

Returns: `IsManiplex`

Constructs the maniplex from the given poset *P*. This assumes that P actually defines a maniplex.

3.1.6 IsPolytopal (for IsManiplex)

▷ `IsPolytopal(M)` (property)

Returns: `true` or `false`

Returns whether the maniplex *M* is polytopal; i.e., the flag graph of a polytope.

3.2 Mixing of Maniplexes functions

3.2.1 Mix (for IsPermGroup, IsPermGroup)

▷ Mix(*permgroup, permgroup*) (operation)

Returns: IsGroup .

Given two (permutation) groups returns the mix of those groups. Note, also works with FPgroups. Here we build the mix of the connection groups of a 3-cube and an edge.

Example

```
gap> g1:=ConnectionGroup(Cube(3));
<permutation group with 3 generators>
gap> g2:=ConnectionGroup(Edge());
Group([ (1,2) ])
gap> Mix(g1,g2);
<permutation group with 3 generators>
```

3.2.2 Mix (for IsFpGroup, IsFpGroup)

▷ Mix(*fpgroup, fpgroup*) (operation)

Returns the Mix of two Finitely Presented groups gp and gq.

3.2.3 Mix (for IsReflexibleManiplex, IsReflexibleManiplex)

▷ Mix(*maniplex, maniplex*) (operation)

Returns: IsReflexibleManiplex .

Given maniplexes returns the IsReflexibleManiplex from the mix of their connection groups

3.2.4 Comix (for IsFpGroup, IsFpGroup)

▷ Comix(*fpgroup, fpgroup*) (operation)

Returns the comix of two Finitely Presented groups gp and gq.

3.2.5 Comix (for IsReflexibleManiplex, IsReflexibleManiplex)

▷ Comix(*maniplex, maniplex*) (operation)

Returns: IsReflexibleManiplex .

Given maniplexes returns the IsReflexibleManiplex from the comix of their connection groups

3.2.6 CtoL (for IsInt,IsInt,IsInt,IsInt)

▷ CtoL(*int, int, int, int*) (operation)

Returns: IsInteger .

CtoL Returns an integer between 1 and N*M associated with the pair [a,b]. LtoC Returns an ordered pair [a,b] associated with the integer between 1 and N*M. a should range between 1 and N,

and b should range between 1 and M N is how many columns (x coordinates), M is how many rows (y coordinates) in a matrix Functions are inverses.

3.2.7 FlagMix (for IsManiplex, IsManiplex)

▷ FlagMix(*permgroup, permgroup*) (operation)

Returns: IsManiplex .

Given two (permutation) groups gp, gg this returns the maniplex of the "flag" mix of two maniplexes with connection groups gp and gq.

3.3 Rotary maniplexes and rotation groups

3.3.1 UniversalRotationGroup (for IsInt)

▷ UniversalRotationGroup(*n*) (operation)

Returns the rotation subgroup of the universal Coxeter Group of rank n.

3.3.2 UniversalRotationGroup (for IsList)

▷ UniversalRotationGroup(*sym*) (operation)

Returns the rotation subgroup of the Coxeter Group with Schlafli symbol sym.

3.3.3 RotaryManiplex (for IsGroup)

▷ RotaryManiplex(*g*) (operation)

Given a group g (which should be a string rotation group), returns the rotary maniplex with that rotation group, where the privileged generators are those returned by GeneratorsOfGroup(g).

3.3.4 RotaryManiplex (for IsList)

▷ RotaryManiplex(*sym*) (operation)

Returns the universal rotary maniplex (in fact, regular polytope) with Schlafli symbol *sym*.

3.3.5 RotaryManiplex (for IsList, IsList)

▷ RotaryManiplex(*symbol, relations*) (operation)

Returns the rotary maniplex with the given Schlafli symbol and with the given relations. The relations are given by a string that refers to the generators s1, s2, etc. For example:

Example

```
gap> M := RotaryManiplex([4,4], "(s2^-1 s1)^6");;
```

If the option set_schlafli is set, then we set the Schlafli symbol to the one given. This may not be the correct Schlafli symbol, since the relations may cause a collapse, so this should only be used if you know that the Schlafli symbol is correct.

3.3.6 EnantiomorphicForm (for IsRotaryManiplex)

▷ `EnantiomorphicForm(`*M*`)` (operation)

The *enantiomorphic form* of a rotary maniplex is the same maniplex, but where we choose the new base flag to be one of the flags that is adjacent to the original base flag. If M is reflexible, then this choice has no effect. Otherwise, if M is chiral, then the enantiomorphic form gives us a different presentation for the rotation group.

Chapter 4

Maniplex Properties

4.1 Automorphism group acting on faces and chains

4.1.1 AutomorphismGroupOnChains (for IsManiplex, IsCollection)

▷ `AutomorphismGroupOnChains(M, I)` (operation)

Returns: IsPermGroup

Returns a permutation group, representing the action of AutomorphismGroup(M) on the chains of M of type I.

Example

```
gap> AutomorphismGroupOnChains(HemiCube(3),[0,2]);
Group([ (1,2)(3,4)(5,10)(6,9)(7,8)(11,12), (2,6)(3,5)(4,7)(8,11)(10,12), (1,3)(2,4)(6,11)(7,8)
  (9,12) ])
```

4.1.2 AutomorphismGroupOnIFaces (for IsManiplex, IsInt)

▷ `AutomorphismGroupOnIFaces(M, i)` (operation)

Returns: IsPermGroup

Returns a permutation group, representing the action of AutomorphismGroup(M) on the i-faces of M.

Example

```
gap> AutomorphismGroupOnIFaces(HemiCube(3),2);
Group([ (), (2,3), (1,2) ])
```

4.1.3 AutomorphismGroupOnVertices (for IsManiplex)

▷ `AutomorphismGroupOnVertices(M)` (attribute)

Returns: IsPermGroup

Returns a permutation group, representing the action of AutomorphismGroup(M) on the vertices of M.

Example

```
gap> AutomorphismGroupOnVertices(HemiCube(4));
Group([ (1,2)(3,4)(5,6)(7,8), (2,3)(6,8), (3,5)(4,6), (5,7)(6,8) ])
```

4.1.4 AutomorphismGroupOnEdges (for IsManiplex)

▷ AutomorphismGroupOnEdges(*M*) (attribute)

Returns: IsPermGroup

Returns a permutation group, representing the action of AutomorphismGroup(*M*) on the edges of *M*.

Example

```
gap> AutomorphismGroupOnEdges(Simplex(4));
Group([ (2,5)(3,6)(4,7), (1,2)(6,8)(7,9), (2,3)(5,6)(9,10), (3,4)(6,7)(8,9) ])
```

4.1.5 AutomorphismGroupOnFacets (for IsManiplex)

▷ AutomorphismGroupOnFacets(*M*) (attribute)

Returns: IsPermGroup

Returns a permutation group, representing the action of AutomorphismGroup(*M*) on the facets of *M*.

Example

```
gap> AutomorphismGroupOnFacets(HemiCube(5));
Group([ (), (4,5), (3,4), (2,3), (1,2) ])
```

4.2 Number of orbits and transitivity

4.2.1 NumberOfChainOrbits (for IsManiplex, IsCollection)

▷ NumberOfChainOrbits(*M*, *I*) (operation)

Returns: IsInt

Returns the number of orbits of chains of type *I* under the action of AutomorphismGroup(*M*).

Example

```
gap> NumberOfChainOrbits(Cuboctahedron(),[0,2]);
2
```

4.2.2 NumberOfIFaceOrbits (for IsManiplex, IsInt)

▷ NumberOfIFaceOrbits(*M*, *i*) (operation)

Returns: IsInt

Returns the number of orbits of *i*-faces under the action of AutomorphismGroup(*M*).

Example

```
gap> NumberOfIFaceOrbits(SnubDodecahedron(),2);
3
```

4.2.3 NumberOfVertexOrbits (for IsManiplex)

▷ NumberOfVertexOrbits(*M*) (attribute)

Returns: IsInt

Returns the number of orbits of vertices under the action of AutomorphismGroup(*M*).

Example

```
gap> NumberOfVertexOrbits(Dual(SnubDodecahedron()));
3
```

4.2.4 NumberOfEdgeOrbits (for IsManiplex)

▷ NumberOfEdgeOrbits(*M*) (attribute)

Returns: IsInt

Returns the number of orbits of edges under the action of AutomorphismGroup(*M*).

Example

```
gap> NumberOfEdgeOrbits(SnubDodecahedron());
3
```

4.2.5 NumberOfFacetOrbits (for IsManiplex)

▷ NumberOfFacetOrbits(*M*) (attribute)

Returns: IsInt

Returns the number of orbits of facets under the action of AutomorphismGroup(*M*).

Example

```
gap> NumberOfFacetOrbits(SnubCube());
3
```

4.2.6 IsChainTransitive (for IsManiplex, IsCollection)

▷ IsChainTransitive(*M*, *I*) (operation)

Returns: IsBool

Determines whether the action of AutomorphismGroup(*M*) on chains of type *I* is transitive.

Example

```
gap> IsChainTransitive(SmallRhombicuboctahedron(),[0,2]);
false
gap> IsChainTransitive(SmallRhombicuboctahedron(),[0,1]);
false
gap> IsChainTransitive(Cuboctahedron(),[0,1]);
true
```

4.2.7 IsIFaceTransitive (for IsManiplex, IsInt)

▷ IsIFaceTransitive(*M*, *i*) (operation)

Returns: IsBool

Determines whether the action of AutomorphismGroup(*M*) on *i*-faces is transitive.

Example

```
gap> IsIFaceTransitive(Cuboctahedron(),1);
true
```

4.2.8 IsVertexTransitive (for IsManiplex)

▷ IsVertexTransitive(*M*) (property)

Returns: IsBool

Determines whether the action of AutomorphismGroup(*M*) on vertices is transitive.

Example

```
gap> IsVertexTransitive(Bk2l(4,5));
true
```

4.2.9 IsEdgeTransitive (for IsManiplex)

▷ IsEdgeTransitive(*M*) (property)

Returns: IsBool

Determines whether the action of AutomorphismGroup(*M*) on edges is transitive.

Example

```
gap> IsEdgeTransitive(Prism(Simplex(3)));
false
```

4.2.10 IsFacetTransitive (for IsManiplex)

▷ IsFacetTransitive(*M*) (property)

Returns: IsBool

Determines whether the action of AutomorphismGroup(*M*) on facets is transitive.

Example

```
gap> IsFacetTransitive(Prism(HemiCube(3)));
false
```

4.2.11 IsFullyTransitive (for IsManiplex)

▷ IsFullyTransitive(*M*) (property)

Returns: IsBool

Determines whether the action of AutomorphismGroup(*M*) on i-faces is transitive for every i.

Example

```
gap> IsFullyTransitive(SmallRhombicuboctahedron());
false
gap> IsFullyTransitive(Bk2l(4,5));
true
```

4.3 Flag orbits

4.3.1 SymmetryTypeGraph (for IsManiplex)

▷ SymmetryTypeGraph(*M*) (attribute)

Returns the Symmetry Type Graph of the maniplex *M*, encoded as a permutation group on Rank(*M*) generators.

Example

```
gap> SymmetryTypeGraph(Prism(Simplex(3)));
Edge labeled graph with 4 vertices, and edge labels [ 0, 1, 2, 3 ]
```

4.3.2 NumberOfFlagOrbits (for IsManiplex)

▷ NumberOfFlagOrbits(*M*) (attribute)

Returns the number of orbits of the automorphism group of *M* on its flags.

Example

```
gap> NumberOfFlagOrbits(Prism(Simplex(3)));
4
```

4.3.3 FlagOrbitRepresentatives (for IsManiplex)

▷ FlagOrbitRepresentatives(*M*) (attribute)

Returns one flag from each orbit under the action of AutomorphismGroup(*M*).

Example

```
gap> FlagOrbitRepresentatives(Prism(Simplex(3)));
[ 1, 49, 97, 145 ]
```

4.3.4 FlagOrbitsStabilizer (for IsManiplex)

▷ FlagOrbitsStabilizer(*M*) (attribute)

Returns: g

Returns the subgroup of the connection group that preserves the flag orbits under the action of the automorphism group.

Example

```
gap> m:=Prism(Dodecahedron());
Prism(Dodecahedron())
gap> s:=FlagOrbitsStabilizer(m);
<permutation group of size 207360000 with 12 generators>
gap> IsSubgroup(ConnectionGroup(m),s);
true
gap> AsSet(Orbit(AutomorphismGroupOnFlags(m),1))=AsSet(Orbit(s,1));
true
```

4.3.5 IsReflexible (for IsManiplex)

▷ IsReflexible(*M*) (property)

Returns: Whether the maniplex *M* is reflexible (has one flag orbit).

Example

```
gap> IsReflexible(Epsilonk(6));
true
```

4.3.6 IsChiral (for IsManiplex)

▷ IsChiral(*M*) (property)

Returns: Whether the maniplex *M* is chrial.

Example

```
gap> IsChiral(ToroidalMap44([2,3]));
true
```

4.3.7 IsRotary (for IsManiplex)

▷ IsRotary(*M*) (property)

Returns: Whether the maniplex *M* is rotary; i.e., whether it is either reflexible or chiral.

Example

```
gap> IsRotary(ToroidalMap44([3,5]));
true
```

4.3.8 FlagOrbits (for IsManiplex)

▷ FlagOrbits(*M*) (attribute)

Returns a list of lists of flags, representing the orbits of flags under the action of AutomorphismGroup(*M*).

Example

```
gap> FlagOrbits(ToroidalMap44([3,2]));
[ [ 1, 9, 7, 33, 15, 63, 5, 65, 39, 23, 13, 71, 61, 101, 3, 89, 47, 37, 95, 21, 11, 79, 69, 29, 5
    [ 2, 10, 8, 34, 16, 64, 6, 66, 40, 24, 14, 72, 62, 102, 4, 90, 48, 38, 96, 22, 12, 80, 70, 30,
```

4.4 Orientability

4.4.1 IsOrientable (for IsManiplex)

▷ IsOrientable(*M*) (property)

Returns: true or false

A maniplex is orientable if its flag graph is bipartite.

Example

```
gap> IsOrientable(HemiCube(3));
false
gap> IsOrientable(Cube(3));
true
```

4.4.2 IsIOrientable (for IsManiplex, IsList)

▷ IsIOrientable(*M, I*) (operation)

For a subset I of {0, ..., n-1}, a maniplex is I-orientable if every closed path in its flag graph contains an even number of edges with colors in I.

Example

```
gap> IsIOrientable(HemiCube(3),[1,2]);
true
```

4.4.3 IsVertexBipartite (for IsManiplex)

▷ IsVertexBipartite(*M*) (property)

Returns: true or false

A maniplex is vertex-bipartite if its 1-skeleton is bipartite. This is equivalent to being I-orientable for I = {0}.

Example

```
gap> IsVertexBipartite(HemiCube(4));
true
```

4.4.4 IsFacetBipartite (for IsManiplex)

▷ IsFacetBipartite(*M*) (property)

Returns: true or false

A maniplex is facet-bipartite if the 1-skeleton of its dual is bipartite. This is equivalent to being I-orientable for I = {n-1}.

Example

```
gap> IsFacetBipartite(HemiCube(4));
false
```

4.4.5 OrientableCover (for IsManiplex)

▷ OrientableCover(*M*) (attribute)

Returns the minimal *orientable cover* of the maniplex *M*.

Example

```
gap> OrientableCover(HemiCube(3))=Cube(3);
true
```

4.4.6 IOrientableCover (for IsManiplex, IsList)

▷ IOrientableCover(*M*) (operation)

Returns the minimal *I-orientable cover* of the maniplex *M*.

4.5 Faithfulness

4.5.1 IsVertexFaithful (for IsReflexibleManiplex)

▷ IsVertexFaithful(*M*) (property)

Returns: true or false

Returns whether the reflexible maniplex *M* is vertex-faithful; i.e., whether the action of the automorphism group on the vertices is faithful.

Example

```
gap> IsVertexFaithful(HemiCube(3));
true
```

4.5.2 IsFacetFaithful (for IsReflexibleManiplex)

▷ IsFacetFaithful(*M*) (property)

Returns: true or false

Returns whether the reflexible maniplex *M* is facet-faithful; i.e., whether the action of the automorphism group on the facets is faithful.

Example

```
gap> IsFacetFaithful(HemiCube(3));
false
gap> IsFacetFaithful(Cube(3));
true
```

4.5.3 MaxVertexFaithfulQuotient (for IsReflexibleManiplex)

▷ `MaxVertexFaithfulQuotient(M)` (operation)

Returns the maximal vertex-faithful reflexible maniplex covered by M.

Example

```
gap> MaxVertexFaithfulQuotient(HemiCrossPolytope(3));
reflexible 3-maniplex
gap> SchlafliSymbol(last);
[ 3, 2 ]
```

Chapter 5

Combinatorics and Structure

5.1 Faces

5.1.1 NumberOfIFaces (for IsManiplex, IsInt)

▷ `NumberOfIFaces(M, i)` (operation)

Returns The number of i-faces of M.

Example

```
gap> NumberOfIFaces(Dodecahedron(),1);
30
```

5.1.2 NumberOfVertices (for IsManiplex)

▷ `NumberOfVertices(M)` (attribute)

Returns the number of vertices of M.

Example

```
gap> NumberOfVertices(HemiDodecahedron());
10
```

5.1.3 NumberOfEdges (for IsManiplex)

▷ `NumberOfEdges(M)` (attribute)

Returns the number of edges of M.

Example

```
gap> NumberOfEdges(HemiIcosahedron());
15
```

5.1.4 NumberOfFacets (for IsManiplex)

▷ `NumberOfFacets(M)` (attribute)

Returns the number of facets of M.

Example

```
gap> NumberOfFacets(Bk2l(4,6));
4
```

5.1.5 NumberOfRidges (for IsManiplex)

▷ `NumberOfRidges(M)` (attribute)

Returns the number of ridges ((n-2)-faces) of *M*.

Example

```
gap> NumberOfRidges(CrossPolytope(5));
80
```

5.1.6 Fvector (for IsManiplex)

▷ `Fvector(M)` (attribute)

Returns the f-vector of *M*.

Example

```
gap> Fvector(HemiIcosahedron());
[ 6, 15, 10 ]
```

5.1.7 Section (for IsManiplex, IsInt, IsInt)

▷ `Section(M, j, i)` (operation)

Returns the section F_j / F_i, where F_j is the j-face of the base flag of *M* and F_i is the i-face of the base flag.

5.1.8 Section (for IsManiplex, IsInt, IsInt, IsInt)

▷ `Section(M, j, i, k)` (operation)

Returns the section F_j / F_i, where F_j is the j-face of flag number *k* of *M* and F_i is the i-face of the same flag.

5.1.9 Sections (for IsManiplex, IsInt, IsInt)

▷ `Sections(M, j, i)` (operation)

Returns all sections of type F_j / F_i, where F_j is a j-face and F_i is an incident i-face.

5.1.10 Facets (for IsManiplex)

▷ `Facets(M)` (attribute)

Returns the facet-types of *M* (i.e. the maniplexes corresponding to the facets).

5.1.11 Facet (for IsManiplex, IsInt)

▷ `Facet(M, k)` (operation)

Returns the facet of `M` that contains the flag number `k` (that is, the maniplex corresponding to the facet).

5.1.12 Facet (for IsManiplex)

▷ `Facet(M)` (attribute)

Returns the facet of `M` that contains flag number 1 (that is, the maniplex corresponding to the facet).

5.1.13 VertexFigures (for IsManiplex)

▷ `VertexFigures(M)` (attribute)

Returns the types of vertex-figures of `M` (i.e. the maniplexes corresponding to the vertex-figures).

5.1.14 VertexFigure (for IsManiplex, IsInt)

▷ `VertexFigure(M, k)` (operation)

Returns the vertex-figure of `M` that contains flag number `k`.

5.1.15 VertexFigure (for IsManiplex)

▷ `VertexFigure(M)` (attribute)

Returns the vertex-figure of `M` that contains the base flag.

5.2 Flatness

5.2.1 IsFlat

▷ `IsFlat(M)` (property)
▷ `IsFlat(M, i, j)` (operation)

Returns: `true` or `false`

In the first form, returns true if every vertex of the maniplex `M` is incident to every facet. In the second form, returns true if every i-face of the maniplex `M` is incident to every j-face.

5.3 Schlafli symbol

5.3.1 SchlafliSymbol (for IsManiplex)

▷ `SchlafliSymbol(M)` (attribute)

Returns the Schlafli symbol of the maniplex `M`. Each entry is either an integer or a set of integers, where entry number i shows the polygons that we obtain as sections of (i+1)-faces over (i-2)-faces.

5.3.2 PseudoSchlafliSymbol (for IsManiplex)

▷ `PseudoSchlafliSymbol(M)` (attribute)

Sometimes when we make a maniplex, we know that the Schlafli symbol must be a quotient of some symbol. This most frequently happens because we start with a maniplex with a given Schlafli symbol and then take a quotient of it. In this case, we store the given Schlafli symbol and call it a *pseudo-Schlafli symbol* of `M`. Note that whenever we compute the actual Schlafli symbol of `M`, we update the pseudo-Schlafli symbol to match.

5.3.3 IsEquivelar (for IsManiplex)

▷ `IsEquivelar(M)` (property)

Returns: the the maniplex `M` is equivelar; i.e., whether its Schlafli Symbol consists of integers at each position (no lists).

5.3.4 IsDegenerate (for IsManiplex)

▷ `IsDegenerate(M)` (property)

Returns: `true` or `false`

Returns whether the maniplex `M` has any sections that are digons. We may eventually want to include maniplexes with even smaller sections.

5.3.5 IsTight (for IsManiplex)

▷ `IsTight(P)` (property)

Returns: `true` or `false`

Returns whether the polytope `P` is tight, meaning that it has a Schlafli symbol {k_1, ..., k_{n-1}} and has 2 k_1 ... k_{n-1} flags, which is the minimum possible. This property doesn't make any sense for non-polytopal maniplexes, which aren't constrained by this lower bound.

5.3.6 EulerCharacteristic (for IsManiplex)

▷ `EulerCharacteristic(M)` (attribute)

Returns: The Euler characteristic of the maniplex, given by $f_0 - f_1 + f_2 - \cdots + (-1)^{n-1} f_{n-1}$.

5.3.7 Genus (for IsManiplex)

▷ `Genus(M)` (attribute)

Returns: The genus of the given 3-maniplex.

5.3.8 IsSpherical (for IsManiplex)

▷ IsSpherical(*M*) (property)

Returns: Whether the 3-maniplex *M* is spherical, which is to say, whether the Euler characteristic is equal to 2.

Example

```
gap> IsSpherical(Simplex(3));
true
gap> IsSpherical(AbstractRegularPolytope([4,4],"h2^3"));
false
gap> IsSpherical(Pyramid(5));
true
gap> IsSpherical(CubicTiling(2));
false
```

5.3.9 IsLocallySpherical (for IsManiplex)

▷ IsLocallySpherical(*M*) (property)

Returns: Whether the 4-maniplex *M* is locally spherical, which is to say, whether its facets and vertex-figures are both spherical.

Example

```
gap> IsLocallySpherical(Simplex(4));
true
gap> IsLocallySpherical(AbstractRegularPolytope([4,4,4]));
false
gap> IsLocallySpherical(CubicTiling(3));
true
gap> IsLocallySpherical(Pyramid(Cube(3)));
true
```

5.3.10 IsToroidal (for IsManiplex)

▷ IsToroidal(*M*) (property)

Returns: Whether the 3-maniplex *M* is toroidal, which is to say, whether the Euler characteristic is equal to 0.

Example

```
gap> IsToroidal(Simplex(3));
false
gap> IsToroidal(AbstractRegularPolytope([4,4],"h2^3"));
true
gap> IsToroidal(Pyramid(5));
false
```

5.3.11 IsLocallyToroidal (for IsManiplex)

▷ IsLocallyToroidal(*M*) (property)

Returns: Whether the 4-maniplex *M* is locally toroidal, which is to say, whether it has at least one toroidal facet or vertex-figure, and all of its facets and vertex-figures are either spherical or toroidal.

Example

```
gap> IsLocallyToroidal(Simplex(4));
false
gap> IsLocallyToroidal(AbstractRegularPolytope([4,4,3],"(r0 r1 r2 r1)^2"));
true
gap> IsLocallyToroidal(AbstractRegularPolytope([4,4,4],"(r0 r1 r2 r1)^2, (r1 r2 r3 r2)^2"));
true
```

5.4 Basics

5.4.1 Size (for IsManiplex)

▷ `Size(M)` (attribute)

Returns: The number of flags of the maniplex *M*.

Synonym: `NumberOfFlags`.

5.4.2 RankManiplex (for IsManiplex)

▷ `RankManiplex(M)` (attribute)

Returns: The rank of the maniplex *M*.

5.5 Zigzags and holes

5.5.1 ZigzagLength (for IsManiplex, IsInt)

▷ `ZigzagLength(M, j)` (operation)

Returns: The lengths of *j*-zigzags of the 3-maniplex *M*. This corresponds to the lengths of orbits under r0 (r1 r2)^j.

5.5.2 ZigzagVector (for IsManiplex)

▷ `ZigzagVector(M)` (attribute)

Returns: The lengths of all zigzags of the 3-maniplex *M*. A rank 3 maniplex of type {p, q} has Floor(q/2) distinct zigzag lengths because the j-zigzags are the same as the (q-j)-zigzags.

5.5.3 PetrieLength (for IsManiplex)

▷ `PetrieLength(M)` (attribute)

Returns: The length of the petrie polygons of the maniplex *M*.

5.5.4 HoleLength (for IsManiplex, IsInt)

▷ `HoleLength(M, j)` (operation)

Returns: The lengths of *j*-holes of the 3-maniplex *M*. This corresponds to the lengths of orbits under r0 (r1 r2)^(j-1) r2.

5.5.5 HoleVector (for IsManiplex)

▷ `HoleVector(M)` (attribute)

Returns: The lengths of all zigzags of the 3-maniplex `M`. A rank 3 maniplex of type {p, q} has Floor(q/2) distinct zigzag lengths because the j-zigzags are the same as the (q-j)-zigzags.

Chapter 6

Comparing maniplexes

6.1 Quotients and covers

Many of the quotient operations let you describe some relations in terms of a string. We assume that Sggis have a generating set of $\{r0, r1, ..., r_{n-1}\}$, so these relation strings will look something like "(r0 r1 r2)^5, r2 = (r0 r1)^3". Notice that we can mix relations like "r2 = (r0 r1)^3" with relators like "(r0 r1 r2)^5"; the latter is treated as the relation "(r0 r1 r2)^5 = 1". For convenience, we also allow relations to contain the following strings: s1, s2, s3, etc, where si is expanded to r(i-1) ri. For example, s2 becomes r1 r2. z1, z2, z3, etc, where zi is expanded to r0 (r1 r2)^i (the "i-zigzag" word). h1, h2, h3, etc, where hi is expanded to r0 (r1 r2)^(j-1) r1 (the "i-hole" word). We note that these strings are all restricted to have $i \leq 9$, *including ri*. This restriction might be changed eventually, but it will require a rewrite of the method ParseStringCRels that underlies many quotient operations.

6.1.1 IsQuotient (for IsManiplex, IsManiplex)

▷ `IsQuotient(M1, M2)` (operation)

Returns whether `M2` is a quotient of `M1`.

6.1.2 IsQuotient (for IsSggi, IsSggi)

▷ `IsQuotient(g, h)` (operation)

Returns whether `h` is a quotient of `g`. That is, whether there is a homomorphism sending each generator of g to the corresponding generator of h.

6.1.3 IsCover (for IsManiplex, IsManiplex)

▷ `IsCover(M1, M2)` (operation)

Returns whether `M2` is a cover of `M1`.

6.1.4 IsIsomorphicManiplex (for IsManiplex, IsManiplex)

▷ IsIsomorphicManiplex(*M1, M2*) (operation)

Returns whether *M1* is isomorphic to *M2*.

6.1.5 SmallestReflexibleCover (for IsManiplex)

▷ SmallestReflexibleCover(*M*) (attribute)

Returns the smallest regular cover of *M*, which is the maniplex whose automorphism group is isomorphic to the connection group of *M*.

6.1.6 QuotientManiplex (for IsReflexibleManiplex, IsString)

▷ QuotientManiplex(*M, relStr*) (operation)

Given a reflexible maniplex *M*, generates the subgroup S of AutomorphismGroup(*M*) given by relStr, and returns the quotient maniplex M / S. For example, QuotientManiplex(CubicTiling(2), "(r0 r1 r2 r1)^5, (r1 r0 r1 r2)^2") returns the toroidal map {4,4}_{(5,0),(0,2)}. You can also input this as CubicTiling(2) / "(r0 r1 r2 r1)^5, (r1 r0 r1 r2)^2".

6.1.7 ReflexibleQuotientManiplex (for IsManiplex, IsList)

▷ ReflexibleQuotientManiplex(*M, rels*) (operation)

Given a reflexible maniplex *M*, generates the normal closure N of the subgroup S of AutomorphismGroup(*M*) given by relStr, and returns the quotient maniplex M / N, which will be reflexible. For example, QuotientManiplex(CubicTiling(2), "(r0 r1 r2 r1)^5, (r1 r0 r1 r2)^2") returns the toroidal map {4,4}_{(1,0)}, because the normal closure of the group generated by (r0 r1 r2 r1)^5 and (r1 r0 r1 r2)^2 is the group generated by r0 r1 r2 r1 and r1 r0 r1 r2.

6.1.8 QuotientSggi (for IsGroup, IsList)

▷ QuotientSggi(*g, rels*) (operation)

Returns: the quotient of *g* by *rels*, which is either a list of Tietze words or a string of relations that is parsed by ParseStringCRels.

Example

```
gap> g := UniversalSggi(3);
<fp group of size infinity on the generators [ r0, r1, r2 ]>
gap> h := QuotientSggi(g, "(r0 r1)^5, (r1 r2)^3, (r0 r1 r2)^5");
<fp group on the generators [ r0, r1, r2 ]>
gap> Size(h);
60
```

6.1.9 QuotientSggiByNormalSubgroup (for IsGroup,IsGroup)

▷ QuotientSggiByNormalSubgroup(*g, n*) (operation)

Returns: g/n

Given an sggi *g* and a normal subgroup *n* in *g*, this function will give you the quotient in a way that respects the generators (i.e., the generators of the quotient will be the images of the generators of the original group).

Example

```
gap> g:=AutomorphismGroup(Cube(3));
<fp group of size 48 on the generators [ r0, r1, r2 ]>
gap> q:=QuotientSggiByNormalSubgroup(g,Group([(g.1*g.2*g.3)^3]));
Group([ (1,2)(3,7)(4,6)(5,10)(8,14)(9,16)(11,18)(12,20)(13,17)(15,23)(19,22)(21,24), (1,3)(2,5)(4
gap> Maniplex(q)=HemiCube(3);
true
```

6.1.10 QuotientManiplexByAutomorphismSubgroup (for IsManiplex,IsPermGroup)

▷ `QuotientManiplexByAutomorphismSubgroup(`*m*`, `*h*`)` (operation)

Returns: m/h

Given a maniplex *m*, and a subgroup *h* of the automorphism group on the flags, this function will give you the maniplex in which the orbits of flags under the action of *h* are identified. Note that this function doesn't do any prechecks, and may break easily when `m/h` _isn't_ a maniplex or when `m/h` is of lower rank (sorry!).

Example

```
gap> m:=Cube(3);
Cube(3)
gap> a:=AutomorphismGroupOnFlags(m);
<permutation group with 3 generators>
gap> h:=Group((a.3*a.1*a.2)^3);
Group([ (1,7)(2,3)(4,18)(5,19)(6,20)(8,11)(9,12)(10,13)(14,32)(15,33)(16,34)(17,35)(21,25)(22,26)
gap> q:=QuotientManiplexByAutomorphismSubgroup(m,h);
3-maniplex with 24 flags
gap> last=HemiCube(3);
true
```

Chapter 7

Posets

7.1 Poset constructors

I'm in the process of reconciling all of this, but there are going to be a number of ways to `define` a poset:

- As an `IsPosetOfFlags`, where the underlying description is an ordered list of length $n+2$. Each of the $n+2$ list elements is a list of faces, and the assumption is that these are the faces of rank $i-2$, where i is the index in the master list (e.g., `l[1][1]` would usually correspond to the unique -1 face of a polytope – and there won't be an `l[1][2]`). Each face is then a list of the flags incident with that face.
- As an `IsPosetOfIndices`, where the underlying description is a binary relation on a set of indices, which correspond to labels for the elements of the poset.
- If the poset is known to be atomic, then by a description of the faces in terms of the atoms... usually we'll just need the list of the elements of maximal rank, from which all other elements may be obtained.
- As an `IsPosetOfElements`, where the elements could be anything, and we have a known function determining the partial order on the elements.

Usually, we assume that the poset will have a natural rank function on it. More information on the poset attributes that are important in the study of abstract polytopes and maniplexes is available in [MS02], [MPW14], and [Wil12].

7.1.1 PosetFromFaceListOfFlags (for IsList)

▷ `PosetFromFaceListOfFlags(`*`list`*`)` (operation)

Returns: `IsPosetOfFlags`.

Given a *`list`* of lists of faces in increasing rank, where each face is described by the incident flags, gives you a IsPosetOfFlags object back. Posets constructed this way are assumed to be IsP1 and IsP2.

Here we have a poset using the `IsPosetOfFlags` description for the triangle.

Example

```
gap> poset:=PosetFromFaceListOfFlags([[[1,2,3,4,5,6]],[[1,2],[3,6],[4,5]],[[1,4],[2,3],[5,6]],[[1
A poset using the IsPosetOfFlags representation with 8 faces.
```

```
gap> FaceListOfPoset(poset);
[ [ [ 1, 2, 3, 4, 5, 6 ] ], [ [ 1, 2 ], [ 3, 6 ], [ 4, 5 ] ], [ [ 1, 4 ], [ 2, 3 ], [ 5, 6 ] ], [
```

7.1.2 PosetFromConnectionGroup (for IsPermGroup)

▷ PosetFromConnectionGroup(*g*) (operation)

Returns: IsPosetOfFlags with IsP1=true.

Given a group, returns a poset with an internal representation as a list of faces ordered by rank, where each face is represented as a list of the flags it contains. Note that this function includes the minimal (empty) face and the maximal face of the maniplex. Note that the i-faces correspond to the $i+1$ item in the list because of how GAP indexes lists.

Example

```
gap> g:=Group([(1,4)(2,3)(5,6),(1,2)(3,6)(4,5)]);
Group([ (1,4)(2,3)(5,6), (1,2)(3,6)(4,5) ])
gap> PosetFromConnectionGroup(g);
A poset using the IsPosetOfFlags representation with 8 faces.
```

7.1.3 PosetFromManiplex (for IsManiplex)

▷ PosetFromManiplex(*mani*) (operation)

Returns: IsPosetOfFlags

Given a maniplex, returns a poset of the maniplex with an internal representation as a list of faces ordered by rank, where each face is represented as a list of the flags it contains. Note that this function does include the minimal (empty) face and the maximal face of the maniplex. Note that the i-faces correspond to the $i+1$ item in the list because of how GAP indexes lists.

Example

```
gap> p:=HemiCube(3);
Regular 3-polytope of type [ 4, 3 ] with 24 flags
gap> PosetFromManiplex(p);
A poset using the IsPosetOfFlags representation with 15 faces.
```

7.1.4 PosetFromPartialOrder (for IsBinaryRelation)

▷ PosetFromPartialOrder(*partialOrder*) (operation)

Returns: IsPosetOfIndices

Given a partial order on a finite set of size n, this function will create a partial order on [1..n].

Example

```
gap> l:=List([[1,1],[1,2],[1,3],[1,4],[2,4],[2,2],[3,3],[4,4]],x->Tuple(x));
gap> r:=BinaryRelationByElements(Domain([1..4]), l);
<general mapping: Domain([ 1 .. 4 ]) -> Domain([ 1 .. 4 ]) >
gap> poset:=PosetFromPartialOrder(r);
A poset using the IsPosetOfIndices representation
gap> h:=HasseDiagramBinaryRelation(PartialOrder(poset));
<general mapping: Domain([ 1 .. 4 ]) -> Domain([ 1 .. 4 ]) >
gap> Successors(h);
[ [ 2, 3 ], [ 4 ], [  ], [  ] ]
```

Note that what we've accomplished here is the poset containing the elements 1, 2, 3, 4 with partial order determined by whether the first element divides the second. The essential information about the poset can be obtained from the Hasse diagram.

7.1.5 PosetFromAtomicList (for IsList)

▷ `PosetFromAtomicList(`*list*`)` (operation)

Returns: `IsPosetOfAtoms`

Given a list of elements, where each element is given as a list of atoms, this function will construct the corresponding poset. Note that this will construct any implied faces as well (i.e., all possible intersections of the listed faces).

Example

```
gap> list:=[[1,2,3],[1,2,4],[1,3,4],[2,3,4]];
[ [ 1, 2, 3 ], [ 1, 2, 4 ], [ 1, 3, 4 ], [ 2, 3, 4 ] ]
gap> poset:=PosetFromAtomicList(list);;
gap> List(Faces(poset),AtomList);
[ [  ], [ 1 ], [ 1, 2 ], [ 1, 2, 3 ], [ 1, 2, 4 ], [ 1, 3 ], [ 1, 3, 4 ], [ 1, 4 ], [ 2 ], [ 2, 3
  [ 2, 3, 4 ], [ 2, 4 ], [ 3 ], [ 3, 4 ], [ 4 ], [ 1 .. 4 ] ]
```

7.1.6 PosetFromElements (for IsList,IsFunction)

▷ `PosetFromElements(`*list_of_faces, func*`)` (operation)

Returns: `IsPosetOfElements`

This is for gathering elements with a known ordering *func* on two variables into a poset. Also note, the expectation is that *func* behaves similarly to IsSubset, i.e., *func* (x,y)=true means y is less than x in the order.

Example

```
gap>  g:=SymmetricGroup(3);
Sym( [ 1 .. 3 ] )
gap> asg:=AllSubgroups(g);
[ Group(()), Group([ (2,3) ]), Group([ (1,2) ]), Group([ (1,3) ]), Group([ (1,2,3) ]),   Group([
gap> poset:=PosetFromElements(asg,IsSubgroup);
A poset on 6 elements using the IsPosetOfIndices representation.
gap> HasseDiagramBinaryRelation(PartialOrder(poset));
<general mapping: Domain([ 1 .. 6 ]) -> Domain([ 1 .. 6 ]) >
gap> Successors(last);
[ [ 2, 3, 4, 5 ], [ 6 ], [ 6 ], [ 6 ], [ 6 ], [  ] ]
gap> List( ElementsList(poset){[2,6]}, ElementObject);
[ Group([ (2,3) ]), Group([ (1,2,3), (2,3) ]) ]
```

7.1.7 PosetFromSuccessorList (for IsList)

▷ `PosetFromSuccessorList(`*successorsList*`)` (operation)

Returns: poset

Given a list of immediate successors, will construct the poset. A valid list of successors is of the form `[[2,3],[3],[]]` where the i-th entry is a list of elements that are greater than the i-th element in the partial order that determines the poset. If the given list isn't reflexive and transitive, this function will induce those properties from the given list of successors.

Example

```
gap> p:=PosetFromManiplex(HemiCube(3));;
gap> Print(p);
PosetFromSuccessorList([ [ 2, 3, 4, 5 ], [ 6, 7, 9 ], [ 6, 8, 11 ], [ 7, 10, 11 ],
[ 8, 9, 10 ], [ 1, 2, 13 ], [ 12, 14 ], [ 12, 14 ], [ 13, 14 ], [ 12, 13 ], [ 13, 14 ],
[ 15 ], [ 15 ], [ 15 ], [ ] ]);
```

7.1.8 Helper functions for special partial orders

▷ PairCompareFlagsList(*list1, list2*) (operation)

▷ PairCompareAtomsList(*list1, list2*) (operation)

Returns: true or false

The functions PairCompareFlagsList and PairCompareAtomsList are used in poset construction. Function assumes *list1* and *list2* are of the form [listOfFlags,i] where listOfFlags is a list of flags in the face and i is the rank of the face. Allows comparison of HasFlagList elements. Function assumes *list1* and *list2* are of the form [listOfAtoms,int] where listOfAtoms is a list of flags in the face and int is the rank of the face. Allows comparison of HasAtomList elements.

7.1.9 DualPoset (for IsPoset)

▷ DualPoset(*poset*) (operation)

Returns: dual

Given a *poset*, will construct a poset isomorphic to the dual of *poset*.

Example

```
gap> p:=PosetFromManiplex(Cube(3));; c:=PosetFromManiplex(CrossPolytope(3));;
gap> IsIsomorphicPoset(DualPoset(DualPoset(p)),p);
true
gap> IsIsomorphicPoset(DualPoset(p),c);
true
gap> IsIsomorphicPoset(DualPoset(p),p);
false
```

7.1.10 Section (for IsFace, IsFace, IsPoset)

▷ Section(*face1, face2, poset*) (operation)

Returns: section

Constructs the section of the *poset face1*/*face2*.

Example

```
gap> poset:=PosetFromManiplex(PyramidOver(Cube(2)));;
gap> faces:=Faces(poset);;List(faces,x->RankInPoset(x,poset));
[ -1, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 2, 2, 2, 2, 2, 3 ]
gap> IsIsomorphicPoset(Section(faces[15],faces[1],poset),PosetFromManiplex(Simplex(2)));
true
gap> IsIsomorphicPoset(Section(faces[16],faces[1],poset),PosetFromManiplex(Cube(2)));
true
gap> IsIsomorphicPoset(Section(faces[20],faces[2],poset),PosetFromManiplex(Cube(2)));
true
```

7.1.11 Cleaving polytopes

▷ `Cleave(p, k)` (operation)
▷ `PartiallyCleave(p, k)` (operation)

Returns: IsPolytope

Given an IsPolytope p, and an IsInt k, `Cleave(polytope,k)` will construct the k^{th}-cleaved polytope of p. Cleaved polytopes were introduced by Daniel Pellicer [Pel18]. `PartiallyCleave(p,k)` will construct the k^{th}-partially cleaved polytope of p.

Example

```
gap> Cleave(PosetFromManiplex(Cube(4)),3);
A poset on 290 elements using the IsPosetOfIndices representation.
```

7.2 Poset attributes

Posets have many properties we might be interested in. Here's a few. All abstract polytope definitions in use here are from Schulte and McMullen's *Abstract Regular Polytopes* [MS02].

7.2.1 MaximalChains (for IsPoset)

▷ `MaximalChains(poset)` (attribute)

Gives the list of maximal chains in a poset in terms of the elements of the poset. Synonyms are `FlagsList` and `Flags`. Tends to work faster (sometimes significantly) if the poset `HasPartialOrder`.

Synonym is `FlagsList`.

Example

```
gap> poset:=PosetFromManiplex(HemiCube(3));
A poset using the IsPosetOfFlags representation.
gap> MaximalChains(poset)[1];
[ An element of a poset made of flags, An element of a poset made of flags,
  An element of a poset made of flags, An element of a poset made of flags,
  An element of a poset made of flags ]
gap> List(last,x->RankInPoset(x,poset));
[ -1, 0, 1, 2, 3 ]
```

7.2.2 RankPoset (for IsPoset)

▷ `RankPoset(poset)` (attribute)

If the poset `IsP1`, ranks are assumed to run from -1 to n, and function will return n. If `IsP1(poset)=false`, ranks are assumed to run from 1 to n. In RAMP, at least currently, we are assuming that graded/ranked posets are bounded. Note that in general what you *actually* want to do is call `Rank(poset)`. The reason is that `Rank` will calculate the `RankPoset` if it isn't set, and then set and store the value in the poset.

7.2.3 ElementsList (for IsPoset)

▷ `ElementsList(`*poset*`)` (attribute)

Will recover the list of faces of the poset, format may depend on *type* of representation of `poset`.

- We also have `FacesList` and `Faces` as synonyms for this command.

7.2.4 OrderingFunction (for IsPoset)

▷ `OrderingFunction(`*poset*`)` (attribute)

`OrderingFunction` is an attribute of a poset which stores a function for ordering elements.

Example

```
gap> p:=PosetFromManiplex(Cube(2));;
gap> p3:=PosetFromElements(RankedFaceListOfPoset(p),PairCompareFlagsList);;
gap> f3:=FacesList(p3);;
gap> OrderingFunction(p3)(ElementObject(f3[2]),ElementObject(f3[1]));
true
gap> OrderingFunction(p3)(ElementObject(f3[1]),ElementObject(f3[2]));
false
```

7.2.5 IsFlaggable (for IsPoset)

▷ `IsFlaggable(`*poset*`)` (property)

Returns: `true` or `false`

Checks or creates the value of the attribute `IsFlaggable` for an `IsPoset`. Point here is to see if the structure of the poset is sufficient to determine the flag graph. For IsPosetOfFlags this is another way of saying that the intersection of the faces (thought of as collections of flags) containing a flag is that selfsame flag. (Might be equivalent to prepolytopal... but Gabe was tired and Gordon hasn't bothered to think about it yet.) Now also works with generic poset element types (not just `IsPosetOfFlags`).

7.2.6 IsAtomic (for IsPoset)

▷ `IsAtomic(`*poset*`)` (property)

Returns: `true` or `false`

This checks whether or not the faces of an IsP1 poset may be described uniquely in terms of the posets atoms.

The terminology as used here is approximately that of Ziegler's *Lectures on Polytopes* where a lattice is atomic if every element is the join of atoms.

Example

```
gap> po:=BinaryRelationOnPoints([[2,3],[4,5],[4,5],[6],[6],[]]);;
gap> po:=ReflexiveClosureBinaryRelation(TransitiveClosureBinaryRelation(po));;
gap> p:=PosetFromPartialOrder(po);; IsAtomic(p);
false
gap> p2:=PosetFromManiplex(Cube(3));; IsAtomic(p2);
true
```

7.2.7 PartialOrder (for IsPoset)

▷ `PartialOrder(`*poset*`)` (attribute)

Returns: `partial order`

HasPartialOrder Checks if *poset* has a declared partial order (binary relation). SetPartialOrder assigns a partial order to the *poset*. In many cases, PartialOrder is able to compute one from structural information.

7.2.8 Lattices

▷ `IsLattice(`*poset*`)` (property)
▷ `IsAllMeets(`*arg*`)` (property)
▷ `IsAllJoins(`*arg*`)` (property)

Returns: IsBool

IsLattice determines whether a poset is a lattice or not. IsAllMeets determines whether all meets in a poset are unique. IsAllJoins determines whether all joins in a poset are unique.

Example

```
gap> poset:=PosetFromManiplex(Cube(3));;
gap> IsLattice(poset);
true
gap> bad:=PosetFromManiplex(HemiCube(3));;
gap> IsLattice(bad);
fail
```

Here's a simple example of when a lattice isn't atomic.

Example

```
gap> l:=[[2,3,4],[5,7],[5,6],[6,7],[8],[8],[8,9],[10],[10],[]];;
gap> b:=BinaryRelationOnPoints(l);;
po:=ReflexiveClosureBinaryRelation(TransitiveClosureBinaryRelation(b));;
gap> poset:=PosetFromPartialOrder(po);;
gap> IsLattice(poset);
true
gap> IsAtomic(poset);
false
```

7.2.9 ListIsP1Poset (for IsList)

▷ `ListIsP1Poset(`*list*`)` (operation)

Returns: `true` or `false`

Given *list*, comprised of sublists of faces ordered by rank, each face listing the flags on the face, this function will tell you if the list corresponds to a P1 poset or not.

7.2.10 IsP1 (for IsPoset)

▷ `IsP1(`*poset*`)` (property)

Returns: `true` or `false`

Determines whether a poset has property P1 from ARP. Recall that a poset is P1 if it has a unique least, and a unique maximal element/face.

Example

```
gap> p:=PosetFromElements(AllSubgroups(AlternatingGroup(4)),IsSubgroup);
A poset using the IsPosetOfIndices representation
gap> IsP1(p);
true
gap> p2:=PosetFromFaceListOfFlags([[[1],[2]],[[1,2]]]);
A poset using the IsPosetOfFlags representation with 3 faces.
gap> IsP1(p2);
false
```

7.2.11 IsP2 (for IsPoset)

▷ IsP2(*poset*) (property)

Returns: true or false

Determines whether a poset has property P2 from ARP. Recall that a poset is P2 if each maximal chain in the poset has the same length (for n-polytopes, this means each flag containes $n+2$ faces).

Example

```
gap> poset:=PosetFromManiplex(HemiCube(3));
gap> IsP2(poset);
true
```

Another nice example

Example

```
gap> g:=AlternatingGroup(4);; a:=AllSubgroups(g);; poset:=PosetFromElements(a,IsSubgroup);
A poset using the IsPosetOfIndices representation
gap> IsP2(poset);
false
```

7.2.12 IsP3 (for IsPoset)

▷ IsP3(*poset*) (property)

Returns: true or false

Determines whether a poset is strongly flag connected (property P3' from ARP). May also be called with command IsStronglyFlagConnected. If you are not working with a pre-polytope, expect this to take a LONG time. This means that given flags Φ and Ψ, not only is there a sequence of flags $\Psi = \Phi_0 = \Phi_1 = \cdots = \Phi_k = \Psi$ such that each Φ_i shares all but once face with Φ_{i+1}, but that each $\Phi_i \supseteq \Phi \cap \Psi$.

Helper for IsP3

7.2.13 IsFlagConnected (for IsPoset)

▷ IsFlagConnected(*poset*) (property)

Returns: true or false

Determines whether a poset is flag connected.

7.2.14 IsP4 (for IsPoset)

▷ IsP4(*poset*) (property)

Returns: true or false

Determines whether a poset satisfies the diamond condition. May also be invoked using IsDiamondCondition. Recall that this means that if F, G elements of the poset of ranks $i-1$ and $i+1$, respectively, where F less than G, then there are precisely two i-faces H such that F is less than H and H is less than G.

7.2.15 IsPolytope (for IsPoset)

▷ IsPolytope(*poset*) (property)

Returns: true or false

Determines whether a poset is an abstract polytope.

Example

```
gap> poset:=PosetFromManiplex(Cube(3));
A poset using the IsPosetOfFlags representation with 28 faces.
gap> IsPolytope(poset);
true
gap> KnownPropertiesOfObject(poset);
[ "IsP1", "IsP2", "IsP3", "IsP4", "IsPolytope" ]
gap> poset2:=PosetFromElements(AllSubgroups(AlternatingGroup(4)),IsSubgroup);
A poset using the IsPosetOfIndices representation
gap> IsPolytope(poset2);
false
gap> KnownPropertiesOfObject(poset2);
[ "IsP1", "IsP2", "IsPolytope" ]
```

7.2.16 IsPrePolytope (for IsPoset)

▷ IsPrePolytope(*poset*) (property)

Returns: true or false

Determines whether a poset is an abstract pre-polytope.

7.2.17 IsSelfDual (for IsPoset)

▷ IsSelfDual(*poset*) (property)

Returns: IsBool

Determines whether a poset is self dual.

Example

```
gap> poset:=PosetFromManiplex(Simplex(5));;
A poset using the IsPosetOfFlags representation.
gap> IsSelfDual(poset);
true
gap> poset2:=PosetFromManiplex(PyramidOver(Cube(3)));;
gap> IsSelfDual(poset2);
false
```

7.3 Working with posets

7.3.1 IsIsomorphicPoset (for IsPoset,IsPoset)

▷ `IsIsomorphicPoset(`*poset1, poset2*`)` (operation)

Returns: `true` or `false`

Determines whether *poset1* and *poset2* are isomorphic by checking to see if their Hasse diagrams are isomorphic.

Example

```
gap> IsIsomorphicPoset( PosetFromManiplex( PyramidOver( Cube(3) ) ),  PosetFromManiplex( PrismOv
false
gap> IsIsomorphicPoset( PosetFromManiplex( PyramidOver( Cube(3) ) ), PosetFromManiplex( PyramidO
true
```

7.3.2 PosetIsomorphism (for IsPoset,IsPoset)

▷ `PosetIsomorphism(`*poset1, poset2*`)` (operation)

Returns: map on face indices

When *poset1* and *poset2* are isomorphic, will give you a map from the faces of *poset1* to the faces of *poset2*.

7.3.3 FlagsAsFlagListFaces (for IsPoset)

▷ `FlagsAsFlagListFaces(`*poset*`)` (operation)

Returns: `IsList`

Given a *poset*, this will give you a version of the list of flags in terms of the proper faces described in the *poset*; i.e., this gives a list of flags where each face is described in terms of its (enumerated) list of incident flags. Note that the flag list does not include the minimal face or the maximal face if the poset IsP2.

7.3.4 RankedFaceListOfPoset (for IsPoset)

▷ `RankedFaceListOfPoset(`*IsPosetOfFlags*`)` (operation)

Returns: `list`

Gives a list of [*face*,*rank*] pairs for all the faces of *poset*. Assumptions here are that faces are lists of incident flags.

7.3.5 AdjacentFlag (for IsPosetOfFlags,IsList,IsInt)

▷ `AdjacentFlag(`*poset, flag, i*`)` (operation)

Returns: `flag(s)`

Given a poset, a flag, and a rank, this function will give you the *i*-adjacent flag. Note that adjacencies are listed from ranks 0 to one less than the dimension. You can replace *flag* with the integer corresponding to that flag. Appending `true` to the arguments will give the position of the flag instead of its description from `FlagsAsFlagListFaces`.

7.3.6 AdjacentFlags (for IsPoset,IsList,IsInt)

▷ AdjacentFlags(*poset, flagaslistoffaces, adjacencyrank*) (operation)

If your poset isn't P4, there may be multiple adjacent maximal chains at a given rank. This function handles that case. May substitute IsInt for flagaslistoffaces corresponding to position of flag in list of maximal chains.

7.3.7 EqualChains (for IsList,IsList)

▷ EqualChains(*flag1, flag2*) (operation)

Determines whether two chains are equal.

7.3.8 ConnectionGeneratorOfPoset (for IsPoset,IsInt)

▷ ConnectionGeneratorOfPoset(*poset, i*) (operation)

Returns: A permutation on the flags.

Given a *poset* and an integer *i*, this function will give you the associated permutation for the rank *i*-connection.

7.3.9 ConnectionGroup (for IsPoset)

▷ ConnectionGroup(*poset*) (attribute)

Returns: IsPermGroup

Given a *poset* that is IsPrePolytope, this function will give you the connection group.

7.3.10 AutomorphismGroup (for IsPoset)

▷ AutomorphismGroup(*poset*) (attribute)

Given a *poset*, gives the automorphism group of the poset as an action on the maximal chains.

7.3.11 AutomorphismGroupOnElements (for IsPoset)

▷ AutomorphismGroupOnElements(*poset*) (attribute)

Given a *poset*, gives the automorphism group of the poset as an action on the elements.

7.3.12 FaceListOfPoset (for IsPoset)

▷ FaceListOfPoset(*poset*) (operation)

Returns: list

Gives a list of faces collected into lists ordered by increasing rank. Suitable as input for PosetFromFaceListOfFlags. Argument must be IsPosetOfFlags.

7.3.13 RankPosetElements (for IsPoset)

▷ `RankPosetElements(`*poset*`)` (operation)

Assigns to each face of a poset (when possible) the rank of the element in the poset.

7.3.14 FacesByRankOfPoset (for IsPoset)

▷ `FacesByRankOfPoset(`*poset*`)` (operation)

Returns: `list`

Gives lists of faces ordered by rank. Also sets the rank for each of the faces.

7.3.15 HasseDiagramOfPoset (for IsPoset)

▷ `HasseDiagramOfPoset(`*poset*`)` (operation)

Returns: directed graph

7.3.16 AsPosetOfAtoms (for IsPoset)

▷ `AsPosetOfAtoms(`*poset*`)` (operation)

Returns: posetFromAtoms

If *poset* is an IsP1 poset admits a description of its elements in terms of its atoms, this function will construct an isomorphic poset whose faces are described using PosetFromAtomList.

Example

```
gap> poset:=PosetFromManiplex(Cube(2));;
gap> p2:=AsPosetOfAtoms(poset);
A poset on 10 elements using the IsPosetOfIndices representation.
gap> IsIsomorphicPoset(poset,p2);
true
```

7.3.17 Max/min faces

▷ `MinFace(`*poset*`)` (operation)

▷ `MaxFace(`*arg*`)` (operation)

Returns: face

Gives the minimal/maximal face of a *poset* when it IsP1 and IsP2.

7.4 Element constructors

7.4.1 PosetElementWithOrder (for IsObject,IsFunction)

▷ `PosetElementWithOrder(`*obj, func*`)` (operation)

Returns: `IsFace`

Creates a `face` with *obj* and ordering function `func`. Note that by convetiontion `func(a,b)` should return true when $b \leq a$.

7.4.2 PosetElementFromListOfFlags (for IsList,IsPoset,IsInt)

▷ PosetElementFromListOfFlags(*list, poset, n*) (operation)

Returns: IsPosetElement

This is used to create a face of rank *n* from a *list* of flags of *poset*.

7.4.3 PosetElementFromAtomList (for IsList)

▷ PosetElementFromAtomList(*list*) (operation)

Returns: IsFace

Creates a face with *list* of atoms. If you wish to assign ranks or membership in a poset, you must do this separately.

7.4.4 PosetElementFromIndex (for IsObject)

▷ PosetElementFromIndex(*obj*) (operation)

Returns: IsFace

Creates a face with index *obj* at rank *n*.

7.4.5 PosetElementWithPartialOrder (for IsObject, IsBinaryRelation)

▷ PosetElementWithPartialOrder(*obj, order*) (operation)

Returns: IsFace

Creates a face with index *obj* and BinaryRelation *order* on *obj*. Function does not check to make sure *order* has *obj* in its domain.

7.4.6 RanksInPosets (for IsPosetElement)

▷ RanksInPosets(*posetelement*) (attribute)

Returns: list

Gives the list of posets *posetelement* is in, and the corresponding rank (if available) as a list of ordered pairs of the form [poset,rank]. #! Note that this attribute is mutable, so if you modify it you may break things.

7.4.7 AddRanksInPosets (for IsPosetElement,IsPoset,IsInt)

▷ AddRanksInPosets(*posetelement, poset, int*) (operation)

Returns: null

Adds an entry in the list of RanksInPosets for *posetelement* corresponding to *poset* with assigned rank *int*.

7.4.8 FlagList (for IsPosetElement)

▷ FlagList(*posetelement, {face}*) (attribute)

Returns: list

Description of *posetelement* n as a list of incident flags (when present).

7.4.9 AtomList (for IsPosetElement)

▷ `AtomList(`*`posetelement, {face}`*`)` (attribute)

Returns: `list`

Description of *`posetelement`* n as a list of atoms (when present).

7.5 Element operations

7.5.1 RankInPoset (for IsPosetElement,IsPoset)

▷ `RankInPoset(`*`[face, poset]`*`)` (operation)

Returns: `IsInt`

Given an element *`face`* and a poset *`poset`* to which it belongs, will give you the rank of *`face`* in *`poset`*.

7.5.2 IsSubface (for IsFace,IsFace,IsPoset)

▷ `IsSubface(`*`[face1, face2, poset]`*`)` (operation)

Returns: `true` or `false`

`face1` and *`face2`* are IsFace or IsPosetElement. IsSubface will check to see if *`face2`* is a subface of *`face1`* in *`poset`*. You may drop the argument *`poset`* if the faces only belong to one poset in common. Warning: if the elements are made up of atoms, then IsSubface doesn't need to know what poset you are working with.

7.5.3 IsEqualFaces (for IsFace, IsFace, IsPoset)

▷ `IsEqualFaces(`*`arg1, arg2, arg3`*`)` (operation)

Determines whether two faces are equal in a poset. Note that \= tests whether they are the identical object or not.

7.5.4 AreIncidentElements (for IsObject,IsObject)

▷ `AreIncidentElements(`*`object1, object2`*`)` (operation)

Returns: `true` or `false`

Given two poset elements, will tell you if they are incident.

- Synonym function: `AreIncidentFaces`.

7.5.5 Meet (for IsFace, IsFace, IsPoset)

▷ `Meet(`*`face1, face2, poset`*`)` (operation)

Returns: meet

Finds (when possible) the meet of two elements in a poset.

7.5.6 Join (for IsFace, IsFace, IsPoset)

▷ Join(*face1, face2, poset*) (operation)

Returns: meet

Finds (when possible) the join of two elements in a poset.

This uses the work of Gleason and Hubard.

7.6 Product operations

The products documented in this section were defined by Gleason and Hubard in [GH18] (https://doi.org/10.1016/j.jcta.2018.02.002).

7.6.1 JoinProduct (for IsPoset,IsPoset)

▷ JoinProduct(*poset1, poset2*) (operation)

Returns: poset

Given two posets, this forms the join product. If given two partial orders, returns the join product of the partial orders. If given two maniplexes, returns the join product of the maniplexes.

Example

```
gap> p:=PosetFromManiplex(Cube(2));
A poset
gap> rel:=BinaryRelationOnPoints([[1,2],[2]]);
Binary Relation on 2 points
gap> p1:=PosetFromPartialOrder(rel);
A poset using the IsPosetOfIndices representation
gap> j:=JoinProduct(p,p1);
A poset using the IsPosetOfIndices representation
gap> IsIsomorphicPoset(j,PosetFromManiplex(PyramidOver(Cube(2))));
true
```

7.6.2 CartesianProduct (for IsPoset,IsPoset)

▷ CartesianProduct(*polytope1, polytope2*) (operation)

Returns: polytope

Given two polytopes, forms the cartesian product of the polytopes. Should also work if you give it any two posets. If given two maniplexes, returns the join product of the maniplexes.

Example

```
gap> p1:=PosetFromManiplex(Edge());
A poset
gap> p2:=PosetFromManiplex(Simplex(2));
A poset
gap> c:=CartesianProduct(p1,p2);
A poset using the IsPosetOfIndices representation
gap> IsIsomorphicPoset(c,PosetFromManiplex(PrismOver(Simplex(2))));
true
```

7.6.3 DirectSumOfPosets (for IsPoset,IsPoset)

▷ DirectSumOfPosets(*polytope1, polytope2*) (operation)

Returns: polytope

Given two polytopes, forms the direct sum of the polytopes.

Example

```
gap> p1:=PosetFromManiplex(Cube(2));;p2:=PosetFromManiplex(Edge());;
gap> ds:=DirectSumOfPosets(p1,p2);
A poset using the IsPosetOfIndices representation.
gap> IsIsomorphicPoset(ds,PosetFromManiplex(CrossPolytope(3)));
true
```

7.6.4 TopologicalProduct (for IsPoset,IsPoset)

▷ TopologicalProduct(*polytope1, polytope2*) (operation)

Returns: polytope

Given two polytopes, forms the topological product of the polytopes. If given two maniplexes, returns the join product of the maniplexes.

Here we demonstrate that the topological product (as expected) when taking the product of a triangle with itself gives us the torus $\{4,4\}_{(3,0)}$ with 72 flags.

Example

```
gap> p:=PosetFromManiplex(Pgon(3));
A poset using the IsPosetOfFlags representation.
gap> tp:=TopologicalProduct(p,p);
A poset using the IsPosetOfIndices representation.
gap> s0 := (5,6);;
gap> s1 := (1,2)(3,5)(4,6);;
gap> s2 := (2,3);;
gap> poly := Group([s0,s1,s2]);;
gap> torus:=PosetFromManiplex(ReflexibleManiplex(poly));
A poset using the IsPosetOfFlags representation.
gap> IsIsomorphicPoset(p,tp);
false
gap> IsIsomorphicPoset(torus,tp);
true
```

7.6.5 Antiprism (for IsPoset)

▷ Antiprism(*polytope*) (operation)

Returns: poset

Given a *polytope* (actually, should work for any poset), will return the antiprism of the *polytope* (poset). If given two maniplexes, returns the join product of the maniplexes.

Example

```
gap> p:=PosetFromManiplex(Pgon(3));;
gap> a:=Antiprism(p);;
gap> IsIsomorphicPoset(a,PosetFromManiplex(CrossPolytope(3)));
true
gap> p:=PosetFromManiplex(Pgon(4));;a:=Antiprism(p);;
gap> d:=DualPoset(p);;ad:=Antiprism(d);;
```

```
gap> IsIsomorphicPoset(a,ad);
true
```

Chapter 8

Polytope Constructions and Operations

8.1 Extensions, amalgamations, and quotients

8.1.1 UniversalPolytope (for IsInt)

▷ `UniversalPolytope(n)` (operation)

Returns: IsManiplex

Constructs the universal polytope of rank `n`.

Example

```
gap> UniversalPolytope(3);
UniversalPolytope(3)
gap> Rank(last);
3
```

8.1.2 UniversalExtension (for IsManiplex)

▷ `UniversalExtension(M)` (operation)

Returns: IsManiplex

Constructs the universal extension of `M`, i.e. the maniplex with facets isomorphic to `M` that covers all other maniplexes with facets isomorphic to `M`. Currently only defined for reflexible maniplexes.

Example

```
gap> UniversalExtension(Cube(3));
regular 4-polytope of type [ 4, 3, infinity ] with infinity flags
```

8.1.3 UniversalExtension (for IsManiplex, IsInt)

▷ `UniversalExtension(M, k)` (operation)

Returns: IsManiplex

Constructs the universal extension of `M` with last entry of Schlafli symbol `k`. Currently only defined for reflexible maniplexes.

Example

```
gap> UniversalExtension(Cube(3),2);
regular 4-polytope of type [ 4, 3, 2 ] with 96 flags
```

8.1.4 TrivialExtension (for IsManiplex)

▷ `TrivialExtension(M)` (operation)

Returns: IsManiplex

Constructs the trivial extension of M, also known as $\{M, 2\}$.

Example

```
gap> TrivialExtension(Dodecahedron());
regular 4-polytope of type [ 5, 3, 2 ] with 240 flags
```

8.1.5 FlatExtension (for IsManiplex, IsInt)

▷ `FlatExtension(M, k)` (operation)

Returns: IsManiplex#! @Description Constructs the flat extension of M with last entry of Schlafli symbol k. (As defined in *Flat Extensions of Abstract Polytopes* [Cun21].)

Currently only defined for reflexible maniplexes.

Example

```
gap> FlatExtension(Simplex(3),4);
reflexible 4-maniplex of type [ 3, 3, 4 ] with 48 flags
```

8.1.6 Amalgamate (for IsManiplex, IsManiplex)

▷ `Amalgamate(M1, M2)` (operation)

Returns: IsManiplex

Constructs the amalgamation of $M1$ and $M2$. Implicitly assumes that $M1$ and $M2$ are compatible. Currently only defined for reflexible maniplexes.

Example

```
gap> Amalgamate(Cube(3),CrossPolytope(3));
reflexible 4-maniplex of type [ 4, 3, 4 ]
```

8.1.7 Medial (for IsManiplex)

▷ `Medial(M)` (operation)

Returns: IsManiplex

Given a 3-maniplex M, returns its medial.

Example

```
gap> SchlafliSymbol(Medial(Dodecahedron()));
[ [ 3, 5 ], 4 ]
```

8.2 Duality

8.2.1 Dual (for IsManiplex)

▷ `Dual(M)` (operation)

Returns: The maniplex that is dual to M.

Example

```
gap> Dual(CrossPolytope(3));
Cube(3)
```

8.2.2 IsSelfDual (for IsManiplex)

▷ IsSelfDual(*M*) (property)

Returns: Whether this maniplex is isomorphic to its dual.

Also works for IsPoset objects.

8.2.3 IsInternallySelfDual (for IsReflexibleManiplex)

▷ IsInternallySelfDual(*M[, x]*) (property)

Returns: true or false

Returns whether this maniplex is "internally self-dual", as defined by Cunningham and Mixer in [CM17] (https://doi.org/10.11575/cdm.v12i2.62785). That is, if *M* is self-dual, and the automorphism of AutomorphismGroup(M) that induces the isomorphism between *M* and its dual is an inner automorphism. If the optional group element *x* is given, then we first check whether *x* is a dualizing automorphism of *M*, and if not, then we search the whole automorphism group of *M*.

Example

```
gap> IsInternallySelfDual(Simplex(4));
true
gap> IsInternallySelfDual(ARP([4,4], "h2^6"));
false
gap> IsInternallySelfDual(ARP([4,4], "h2^5"));
true
gap> IsInternallySelfDual(Cube(3));
false
gap> M := InternallySelfDualPolyhedron2(10,1);;
gap> g := AutomorphismGroup(M);;
gap> IsInternallySelfDual(M, (g.1*g.3*g.2)^6);
true
```

8.2.4 IsExternallySelfDual (for IsReflexibleManiplex)

▷ IsExternallySelfDual(*M*) (property)

Returns: true or false

Returns whether this maniplex is "externally self-dual", as defined by Cunningham and Mixer in [CM17] (https://doi.org/10.11575/cdm.v12i2.62785). That is, if *M* is self-dual, and the automorphism of AutomorphismGroup(M) that induces the isomorphism between *M* and its dual is an outer automorphism.

Example

```
gap> IsExternallySelfDual(Simplex(4));
false
gap> IsExternallySelfDual(ARP([4,4], "h2^6"));
true
gap> IsExternallySelfDual(ARP([4,4], "h2^5"));
false
gap> IsExternallySelfDual(Cube(3));
false
```

8.2.5 Petrial (for IsManiplex)

▷ Petrial(*M*) (attribute)

Returns: The Petrial (Petrie dual) of *M*. Note that this is not necessarily a polytope, even if *M* is. When Rank(M) > 3, this is the "generalized Petrial" which essentially replaces r_{n-3} with $r_{n-3}r_{n-1}$ in the set of generators.

Example

```
gap> Petrial(HemiCube(3));
ReflexibleManiplex([ 3, 3 ], "((r0 r2)*r1*r2)^3,z1^4")
```

8.2.6 IsSelfPetrial (for IsManiplex)

▷ IsSelfPetrial(*M*) (property)

Returns: Whether this maniplex is isomorphic to its Petrial.

Example

```
gap> s0 := ( 2, 3)( 4, 6)( 7,10)( 9,12)(11,14)(13,15);;
gap> s1 := ( 1, 2)( 3, 5)( 4, 7)( 6, 9)( 8,11)(10,13)(12,15)(14,16);;
gap> s2 := ( 2, 4)( 3, 6)( 5, 8)( 9,12)(11,15)(13,14);;
gap> poly := Group([s0,s1,s2]);;
gap> p:=ARP(poly);
regular 3-polytope
gap> IsSelfPetrial(p);
true
```

8.2.7 DirectDerivates (for IsManiplex)

▷ DirectDerivates(*M*) (operation)

Returns a list of the *direct derivates* of *M*, which are the images of M under duality and Petriality. A 3-maniplex has up to 6 direct derivates, and an n-maniplex with $n \geq 4$ has up to 8. If the option 'polytopal' is set, then only returns those direct derivates that are polytopal.

Example

```
gap> DirectDerivates(Cube(3));
[ Cube(3), CrossPolytope(3), ReflexibleManiplex([ 6, 3 ], "z1^4"),
  ReflexibleManiplex([ 6, 4 ], "z1^3"), ReflexibleManiplex([ 3, 6 ], "(r2*r1*r0)^4"),
  ReflexibleManiplex([ 4, 6 ], "(r2*r1*r0)^3") ]
```

8.3 Products

8.3.1 Pyramids

▷ Pyramid(*M*) (operation)

▷ Pyramid(*k*) (operation)

Returns the pyramid over *M*. Returns the pyramid over a *k*-gon.

Example

```
gap> SchlafliSymbol(Pyramid(Cube(3)));
[ [ 3, 4 ], [ 3, 4 ], 3 ]
```

```
gap> SchlafliSymbol(Pyramid(4));
[ [ 3, 4 ], [ 3, 4 ] ]
```

8.3.2 Prisms

▷ Prism(*M*) (operation)
▷ Prism(*k*) (operation)

Returns the prism over *M*. Returns the prism over a *k*-gon.

Example

```
gap> Cube(3)=Prism(Cube(2));
true
gap> Prism(4)=Cube(3);
true
```

8.3.3 Antiprisms

▷ Antiprism(*M*) (operation)
▷ Antiprism(*k*) (operation)

Returns the antiprism over *M*. Returns the antiprism over a *k*-gon.

Example

```
gap> SchlafliSymbol(Antiprism(Dodecahedron()));
[ [ 3, 5 ], [ 3, 5 ], 4 ]
gap> SchlafliSymbol(Antiprism(5));
[ [ 3, 5 ], 4 ]
```

8.3.4 JoinProduct (for IsManiplex, IsManiplex)

▷ JoinProduct(*M1, M2*) (operation)

Returns: Maniplex

Given two maniplexes, this forms the join product. May give weird results if the maniplexes aren't faithfully represented by their posets.

Example

```
gap> SchlafliSymbol(last);
[ [ 3, 4 ], [ 3, 4 ], [ 3, 4 ], [ 3, 4 ], 3 ]
```

8.3.5 CartesianProduct (for IsManiplex, IsManiplex)

▷ CartesianProduct(*M1, M2*) (operation)

Returns: Maniplex

Given two maniplexes, this forms the cartesian product. May give weird results if the maniplexes aren't faithfully represented by their posets.

Example

```
gap> SchlafliSymbol(CartesianProduct(HemiCube(3),Simplex(2)));
[ [ 3, 4 ], 3, 3, 3 ]
```

8.3.6 DirectSumOfManiplexes (for IsManiplex, IsManiplex)

▷ `DirectSumOfManiplexes(`*M1, M2*`)` (operation)

Returns: Maniplex

Given two maniplexes, this forms the direct sum. May give weird results if the maniplexes aren't faithfully represented by their posets.

Example

```
gap> SchlafliSymbol(DirectSumOfManiplexes(HemiCube(3),Simplex(2)));
[ [ 3, 4 ], [ 3, 4 ], [ 3, 4 ], [ 3, 4 ] ]
```

8.3.7 TopologicalProduct (for IsManiplex, IsManiplex)

▷ `TopologicalProduct(`*M1, M2*`)` (operation)

Returns: Maniplex

Given two maniplexes, this forms the direct sum. May give weird results if the maniplexes aren't faithfully represented by their posets.

Example

```
gap> SchlafliSymbol(TopologicalProduct(HemiCube(3),Simplex(2)));
[ 4, 3, [ 3, 4 ] ]
```

Chapter 9

Graphs for Maniplexes

9.1 Graph constructors for maniplexes

9.1.1 DirectedGraphFromListOfEdges (for IsList,IsList)

▷ DirectedGraphFromListOfEdges(*list, list*) (operation)

Returns: IsGraph. Note this returns a directed graph.

Given a list of vertices and a list of directed-edges (represented as ordered pairs), this outputs the directed graph with the appropriate vertex and directed-edge set.

Here we have a directed cycle on 3 vertices.

Example

```
gap> g:= DirectedGraphFromListOfEdges([1,2,3],[[1,2],[2,3],[3,1]]);
rec( adjacencies := [ [ 2 ], [ 3 ], [ 1 ] ], group := Group(()),
 isGraph := true, names := [ 1, 2, 3 ], order := 3,
 representatives := [ 1, 2, 3 ], schreierVector := [ -1, -2, -3 ] )
```

9.1.2 GraphFromListOfEdges (for IsList,IsList)

▷ GraphFromListOfEdges(*list, list*) (operation)

Returns: IsGraph. Note this returns an undirected graph.

Given a list of vertices and a list of (directed) edges (represented as ordered pairs), this outputs the simple underlying graph with the appropriate vertex and directed-edge set.

Here we have a simple complete graph on 4 vertices.

Example

```
gap> g:= GraphFromListOfEdges([1,2,3,4],[[1,2],[2,3],[3,1], [1,4], [2,4], [3,4]]);
rec(
 adjacencies := [ [ 2, 3, 4 ], [ 1, 3, 4 ], [ 1, 2, 4 ], [ 1, 2, 3 ] ],
 group := Group(()), isGraph := true, isSimple := true,
 names := [ 1, 2, 3, 4 ], order := 4, representatives := [ 1, 2, 3, 4 ]
   , schreierVector := [ -1, -2, -3, -4 ] )
```

9.1.3 UnlabeledFlagGraph (for IsGroup)

▷ UnlabeledFlagGraph(*group*) (operation)

Returns: IsGraph. Note this returns an undirected graph.

Given a group (assumed to be the connection group of a maniplex), this outputs the simple underlying flag graph.

Here we build the flag graph for the cube from its connection group.

Example

```
gap> g:= UnlabeledFlagGraph(ConnectionGroup(Cube(3)));
rec(
 adjacencies := [ [ 3, 11, 20 ], [ 7, 13, 18 ], [ 1, 4, 10 ],
     [ 3, 25, 34 ], [ 26, 28, 35 ], [ 7, 13, 41 ], [ 2, 6, 8 ],
     [ 7, 27, 32 ], [ 28, 33, 35 ], [ 3, 20, 45 ], [ 1, 14, 23 ],
     [ 15, 17, 24 ], [ 2, 6, 31 ], [ 11, 25, 44 ], [ 12, 45, 47 ],
     [ 18, 28, 40 ], [ 12, 19, 27 ], [ 2, 16, 21 ], [ 17, 22, 24 ],
     [ 1, 10, 38 ], [ 18, 32, 40 ], [ 19, 41, 48 ], [ 11, 35, 44 ],
     [ 12, 19, 34 ], [ 4, 14, 37 ], [ 5, 38, 42 ], [ 8, 17, 30 ],
     [ 5, 9, 16 ], [ 39, 41, 48 ], [ 27, 32, 47 ], [ 13, 33, 39 ],
     [ 8, 21, 30 ], [ 9, 31, 46 ], [ 4, 24, 37 ], [ 5, 9, 23 ],
     [ 43, 45, 47 ], [ 25, 34, 48 ], [ 20, 26, 43 ], [ 29, 31, 46 ],
     [ 16, 21, 42 ], [ 6, 22, 29 ], [ 26, 40, 43 ], [ 36, 38, 42 ],
     [ 14, 23, 46 ], [ 10, 15, 36 ], [ 33, 39, 44 ], [ 15, 30, 36 ],
     [ 22, 29, 37 ] ], group := Group(()), isGraph := true,
 isSimple := true, names := [ 1 .. 48 ], order := 48,
 representatives := [ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14,
     15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30,
     31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46,
     47, 48 ],
 schreierVector := [ -1, -2, -3, -4, -5, -6, -7, -8, -9, -10, -11,
     -12, -13, -14, -15, -16, -17, -18, -19, -20, -21, -22, -23, -24,
     -25, -26, -27, -28, -29, -30, -31, -32, -33, -34, -35, -36, -37,
     -38, -39, -40, -41, -42, -43, -44, -45, -46, -47, -48 ] )
```

This also works with a maniplex input. Here we build the flag graph for the cube.

Example

```
gap> g:= UnlabeledFlagGraph(Cube(3));
```

9.1.4 FlagGraphWithLabels (for IsGroup)

▷ `FlagGraphWithLabels(`*group*`)` (operation)

Returns: a triple [`IsGraph`, `IsList`, `IsList`].

Given a group (assumed to be the connection group of a maniplex), this outputs a triple [graph,list,list]. The graph is the unlabeled flag graph of the connection group. The first list gives the undirected edges in the flag graphs. The second list gives the labels for these edges.

Here we again build the flag graph for the cube from its connection group, but this time keep track of labels of the edges.

Example

```
gap> g:= FlagGraphWithLabels(ConnectionGroup(Cube(3)));
[ rec(
     adjacencies := [ [ 3, 11, 20 ], [ 7, 13, 18 ], [ 1, 4, 10 ],
         [ 3, 25, 34 ], [ 26, 28, 35 ], [ 7, 13, 41 ], [ 2, 6, 8 ],
         [ 7, 27, 32 ], [ 28, 33, 35 ], [ 3, 20, 45 ], [ 1, 14, 23 ],
         [ 15, 17, 24 ], [ 2, 6, 31 ], [ 11, 25, 44 ], [ 12, 45, 47 ],
         [ 18, 28, 40 ], [ 12, 19, 27 ], [ 2, 16, 21 ],
         [ 17, 22, 24 ], [ 1, 10, 38 ], [ 18, 32, 40 ],
```

```
          [ 19, 41, 48 ], [ 11, 35, 44 ], [ 12, 19, 34 ],
          [ 4, 14, 37 ], [ 5, 38, 42 ], [ 8, 17, 30 ], [ 5, 9, 16 ],
          [ 39, 41, 48 ], [ 27, 32, 47 ], [ 13, 33, 39 ],
          [ 8, 21, 30 ], [ 9, 31, 46 ], [ 4, 24, 37 ], [ 5, 9, 23 ],
          [ 43, 45, 47 ], [ 25, 34, 48 ], [ 20, 26, 43 ],
          [ 29, 31, 46 ], [ 16, 21, 42 ], [ 6, 22, 29 ],
          [ 26, 40, 43 ], [ 36, 38, 42 ], [ 14, 23, 46 ],
          [ 10, 15, 36 ], [ 33, 39, 44 ], [ 15, 30, 36 ],
          [ 22, 29, 37 ] ], group := Group(()), isGraph := true,
      isSimple := true, names := [ 1 .. 48 ], order := 48,
      representatives := [ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13,
          14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28,
          29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43,
          44, 45, 46, 47, 48 ],
      schreierVector := [ -1, -2, -3, -4, -5, -6, -7, -8, -9, -10, -11,
          -12, -13, -14, -15, -16, -17, -18, -19, -20, -21, -22, -23,
          -24, -25, -26, -27, -28, -29, -30, -31, -32, -33, -34, -35,
          -36, -37, -38, -39, -40, -41, -42, -43, -44, -45, -46, -47,
          -48 ] ),
  [ [ 1, 3 ], [ 1, 11 ], [ 1, 20 ], [ 2, 7 ], [ 2, 13 ], [ 2, 18 ],
      [ 3, 4 ], [ 3, 10 ], [ 4, 25 ], [ 4, 34 ], [ 5, 26 ], [ 5, 28 ],
      [ 5, 35 ], [ 6, 7 ], [ 6, 13 ], [ 6, 41 ], [ 7, 8 ], [ 8, 27 ],
      [ 8, 32 ], [ 9, 28 ], [ 9, 33 ], [ 9, 35 ], [ 10, 20 ],
      [ 10, 45 ], [ 11, 14 ], [ 11, 23 ], [ 12, 15 ], [ 12, 17 ],
      [ 12, 24 ], [ 13, 31 ], [ 14, 25 ], [ 14, 44 ], [ 15, 45 ],
      [ 15, 47 ], [ 16, 18 ], [ 16, 28 ], [ 16, 40 ], [ 17, 19 ],
      [ 17, 27 ], [ 18, 21 ], [ 19, 22 ], [ 19, 24 ], [ 20, 38 ],
      [ 21, 32 ], [ 21, 40 ], [ 22, 41 ], [ 22, 48 ], [ 23, 35 ],
      [ 23, 44 ], [ 24, 34 ], [ 25, 37 ], [ 26, 38 ], [ 26, 42 ],
      [ 27, 30 ], [ 29, 39 ], [ 29, 41 ], [ 29, 48 ], [ 30, 32 ],
      [ 30, 47 ], [ 31, 33 ], [ 31, 39 ], [ 33, 46 ], [ 34, 37 ],
      [ 36, 43 ], [ 36, 45 ], [ 36, 47 ], [ 37, 48 ], [ 38, 43 ],
      [ 39, 46 ], [ 40, 42 ], [ 42, 43 ], [ 44, 46 ] ],
  [ 3, 2, 1, 3, 1, 2, 2, 1, 3, 1, 2, 3, 1, 1, 3, 2, 2, 1, 3, 1, 2, 3,
      3, 2, 3, 1, 2, 3, 1, 2, 2, 1, 1, 3, 1, 2, 3, 1, 2, 3, 2, 3, 2, 2,
      1, 1, 3, 2, 3, 2, 1, 1, 3, 3, 2, 3, 1, 1, 2, 1, 3, 3, 3, 2, 3, 1,
      2, 3, 1, 2, 1, 2 ] ]
```

This also works with a maniplex input. Here we build the flag graph for the cube.

Example

```
gap> g:= FlagGraphWithLabels(Cube(3));
```

9.1.5 LayerGraph (for IsGroup, IsInt, IsInt)

▷ LayerGraph(*[group, int, int]*) (operation)

Returns: IsGraph. Note this returns an undirected graph.

Given a group (assumed to be the connection group of a maniplex), and two integers, this outputs the simple underlying graph given by incidences of faces of those ranks. Note: There are no warnings yet to make sure that i,j are bounded by the rank.

Here we build the graph given by the 6 faces and 12 edges of a cube from its connection group.

Example

```
gap> g:= LayerGraph(ConnectionGroup(Cube(3)),2,1);
rec(
 adjacencies := [ [ 7, 10, 12, 17 ], [ 8, 10, 15, 18 ],
     [ 7, 9, 13, 14 ], [ 8, 11, 13, 16 ], [ 9, 12, 16, 18 ],
     [ 11, 14, 15, 17 ], [ 1, 3 ], [ 2, 4 ], [ 3, 5 ], [ 1, 2 ],
     [ 4, 6 ], [ 1, 5 ], [ 3, 4 ], [ 3, 6 ], [ 2, 6 ], [ 4, 5 ],
     [ 1, 6 ], [ 2, 5 ] ], group := Group(()), isGraph := true,
 isSimple := true, names := [ 1 .. 18 ], order := 18,
 representatives := [ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14,
     15, 16, 17, 18 ],
 schreierVector := [ -1, -2, -3, -4, -5, -6, -7, -8, -9, -10, -11,
     -12, -13, -14, -15, -16, -17, -18 ] )
```

This also works with a maniplex input. Here we build the graph given by the 6 faces and 12 edges of a cube.

Example

```
gap> g:= LayerGraph(Cube(3),2,1);;
```

9.1.6 Skeleton (for IsManiplex)

▷ `Skeleton(`*maniplex*`)` (operation)

Returns: `IsGraph`. Note this returns an undirected graph.

Given a maniplex, this outputs the 0-1 skeleton. The vertices are the 0-faces, and the edges are the 1-faces.

Here we build the skeleton of the dodecahedron.

Example

```
gap> g:= Skeleton(Dodecahedron());;
```

9.1.7 CoSkeleton (for IsManiplex)

▷ `CoSkeleton(`*maniplex*`)` (operation)

Returns: `IsGraph`. Note this returns an undirected graph.

Given a maniplex, this outputs the $(n-1)$-$(n-2)$ skeleton, i.e., the 0-1 skeleton of the dual. The vertices are the $(n-1)$-faces, and the edges are the $(n-2)$-faces.

Here we build the co-skeleton of the dodecahedron and verify that it is the skeleton of the icosahedron.

Example

```
gap> g:=CoSkeleton(Dodecahedron());;
gap> h:=Skeleton(Icosahedron());;
gap> g=h;
true
```

9.1.8 Hasse (for IsManiplex)

▷ `Hasse(`*group*`)` (operation)

Returns: `IsGraph`. Note this returns a directed graph.

Given a group, assumed to be the connection group of a maniplex, this outputs the Hasse Diagram as a directed graph. Note: The unique minimal and maximal face are assumed.

Here we build the Hasse Diagram of a 3-simplex from its representation as a maniplex.

Example

```
gap> Hasse(Simplex(3));
rec(
 adjacencies := [ [  ], [ 1 ], [ 1 ], [ 1 ], [ 1 ], [ 2, 4 ],
     [ 2, 3 ], [ 3, 5 ], [ 2, 5 ], [ 4, 5 ], [ 3, 4 ], [ 6, 9, 10 ],
     [ 6, 7, 11 ], [ 8, 10, 11 ], [ 7, 8, 9 ], [ 12, 13, 14, 15 ] ],
 group := Group(()), isGraph := true, names := [ 1 .. 16 ],
 order := 16,
 representatives := [ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14,
     15, 16 ],
 schreierVector := [ -1, -2, -3, -4, -5, -6, -7, -8, -9, -10, -11,
     -12, -13, -14, -15, -16 ] )
```

9.1.9 QuotientByLabel (for IsObject,IsList, IsList, IsList)

▷ QuotientByLabel(*object, list, list, list*) (operation)

Returns: IsGraph. Note this returns an undirected graph.

Given a graph, its edges, and its edge labels, and a sublist of labels, this creates the underlying simple graph of the quotient identifying vertices connected by labels not in the sublist.

Here we start with the flag graph of the 3-cube (with edge labels 1,2,3), and identify any vertices not connected by edge by edges of label 1. We can then check that this new graph is bipartite.

Example

```
gap> P:=Cube(3);;
gap> f:=FlagGraphWithLabels(P);;
gap> g:=f[1];;
gap> ed:=f[2];;
gap> lab:=f[3];  #Note This triple is to be replace by a single object.
[ 3, 2, 1, 3, 1, 2, 1, 2, 3, 2, 1, 3, 2, 1, 1, 3, 2, 2, 3, 1, 3, 1, 2, 3, 2, 1, 1, 2, 2, 3, 1, 3,
  3, 1, 2, 1, 3, 2, 2, 1, 2, 2, 3, 1, 1, 3, 1, 3, 3, 2, 1, 2, 1, 3, 3, 1, 3, 2, 2, 2, 2, 3, 3, 1,
gap> Q:=QuotientByLabel(g,ed,lab,[1]);
rec( adjacencies := [ [ 5, 6, 8 ], [ 3, 4, 7 ], [ 2, 6, 8 ], [ 2, 5, 8 ], [ 1, 4, 7 ], [ 1, 3, 7
 isSimple := true, names := [ 1 .. 8 ], order := 8, representatives := [ 1, 2, 3, 4, 5, 6, 7, 8 ]
gap> IsBipartite(Q);
true
```

9.1.10 EdgeLabeledGraphFromEdges (for IsList, IsList,IsList)

▷ EdgeLabeledGraphFromEdges(*list, list, list*) (operation)

Returns: IsEdgeLabeledGraph.

Given a list of vertices, a list of edges, and a list of edge labels, this represents the edge labeled (multi)-graph with those parameters. Semi-edges are represented by a singleton in the edge list. Loops are represented by edges [i,i]

Here we have an edge labeled cycle graph with 6 vertices and edges alternating in labels 0,1.

Example

```
V:=[1..6];;
Edges:=[[1,2],[2,3],[3,4],[4,5],[5,6],[6,1]];;
L:=[0,1,0,1,0,1];;
gamma:=EdgeLabeledGraphFromEdges(V,Edges,L);
```

9.1.11 FlagGraph (for IsGroup)

▷ FlagGraph(*group*) (operation)

Returns: IsEdgeLabeledGraph.

Given group, assumed to be a connection group, output the labeled flag graph. The input could also be a maniplex, then the connection group is calculated.

Here we have the flag graph of the 3-simplex from its connection group.

Example

```
C:=ConnectionGroup(Simplex(3));;
gamma:=FlagGraph(C);
```

9.1.12 UnlabeledSimpleGraph (for IsEdgeLabeledGraph)

▷ UnlabeledSimpleGraph(*edge-labeled-graph*) (operation)

Returns: IsGraph.

Given an edge labeled (multi) graph, it returns the underlying simple graph, with semi-edges, loops, and muliple-edges removed.

Here we have underlying simple graph for the flag graph of the cube.

Example

```
gamma:=UnlabeledSimpleGraph(FlagGraph(Cube(3)));
```

9.1.13 EdgeLabelPreservingAutomorphismGroup (for IsEdgeLabeledGraph)

▷ EdgeLabelPreservingAutomorphismGroup(*edge-labeled-graph*) (operation)

Returns: IsGroup.

Given an edge labeled (multi) graph, it returns automorphism group (preserving the labels). Note, for now the labels are assumed to be [1..n]. Note This tends to be very slow. I would like to look for a way to go back and forth between flag automorphisms and poset automorphisms, as the latter are much faster to compute.

Here we have the automorphism group of the flag graph of the cube.

Example

```
g:=EdgeLabelPreservingAutomorphismGroup(FlagGraph(Cube(3)));;
Size(g);
```

9.1.14 Simple (for IsEdgeLabeledGraph)

▷ Simple(*edge-labeled-graph*) (operation)

Returns: IsEdgeLabeledGraph .

Given an edge labeled (multi) graph, it returns another edge labeled graph where semi-edges, loops, and multiple edges are removed. Note only the "first" edge label is retained if there are multiple edges.

9.1.15 ConnectedComponents (for IsEdgeLabeledGraph, IsList)

▷ ConnectedComponents(*edge-labeled-graph*) (operation)

Returns: IsGraph.

Given an edge labeled (multi) graph and a list of labels, it returns connected components of the graph not using edges in the list of labels. Note if the second argument is not used, it is assumed to be an empty list, and the connected components of the original graph are returned.

Here we see that each connected component of the flag graph of the cube (which has labels 1,2,3) where edges of label 2 are removed, is a 4 cycle.

Example

```
gamma:=ConnectedComponents(FlagGraph(Cube(3)),[2]);
```

9.1.16 PRGraph (for IsGroup)

▷ PRGraph(*group*) (operation)

Returns: IsEdgeLabeledGraph .

Given a group, it returns the permutation representation graph for that group. When the group is a string C-group this is also called a CPR graph. The labels of the edges are [1...r] where r is the number of generators of the group.

Here we see the CPR graph of the automorphism group of a cube (acting on its 8 vertices).

Example

```
G:=AutomorphismGroup(Cube(3));
H:=Group(G.2,G.3);
phi:=FactorCosetAction(G,H);
G2:=Range(phi);
gamma:=PRGraph(G2);
```

9.1.17 CPRGraphFromGroups (for IsGroup,IsGroup)

▷ CPRGraphFromGroups(*group, subgroup*) (operation)

Returns: IsEdgeLabeledGraph.

Given a group and a subgroup. Returns the graph of the action of the first group on cosets of the subgroup.

9.1.18 AdjacentVertices (for IsEdgeLabeledGraph, IsObject)

▷ AdjacentVertices(*EdgeLabeledGraph, vertex*) (operation)

Returns: IsList.

Takes in an edge labeled graph and a vertex, and outputs a list of the adjacent vertices.

9.1.19 LabeledAdjacentVertices (for IsEdgeLabeledGraph, IsObject)

▷ LabeledAdjacentVertices(*EdgeLabeledGraph, vertex*) (operation)

Returns: IsList, IsList.

Takes in an edge labeled graph and a vertex, and outputs two lists: the list of adjacent vertices, and the labels of the corresponding edges.

9.1.20 SemiEdges (for IsEdgeLabeledGraph)

▷ SemiEdges(*EdgeLabeledGraph*) (operation)

Returns: IsList.

Takes in an edge labeled graph and a vertex, and outputs a list of semiedges

9.1.21 LabeledSemiEdges (for IsEdgeLabeledGraph)

▷ LabeledSemiEdges(*EdgeLabeledGraph*) (operation)

Returns: IsList, IsList.

Takes in an edge labeled graph and a vertex, and outputs two lists: SemiEdges and their labels

9.1.22 LabeledDarts (for IsEdgeLabeledGraph)

▷ LabeledDarts(*EdgeLabeledGraph*) (operation)

Returns: IsList.

Takes in an edge labeled graph and outputs the labeled darts.

9.1.23 DerivedGraph (for IsList,IsList,IsList)

▷ DerivedGraph(*list, list, list*) (operation)

Returns: IsEdgeLabeledGraph.

Given a a pre-maniplex (entered as its vertices and labeled darts) and voltages Return the connected derived graph from a pre-maniplex Careful, the order of our automorphisms. Do we want them on left or right? Does it matter? Can make another version with non-connected results, where the group is also an input

Here we can build the flag graph of a 3-orbit polyhedron.

Example

```
gap> V:=[1,2,3];;
gap> Ed:=[[1],[1],[1,2],[2],[2,3],[3],[3]];;
gap> L:=[1,2,0,2,1,0,2];;
gap> g:=EdgeLabeledGraphFromEdges(V,Ed,L);;
gap> L:=LabeledDarts(g);;
gap> volt:=[ (1,2), (3,4), (), (), (3,4), (), (), (4,5), (2,3) ];;
gap> D:=DerivedGraph(V,L,volt);
Edge labeled graph with 360 vertices, and edge labels [ 0, 1, 2 ]
```

9.1.24 ViewGraph (for IsEdgeLabeledGraph, IsString)

▷ ViewGraph(*EdgeLabeledGraph, String*) (operation)

Returns: IsString.

This takes an edge labeled graph and outputs code to view the graph in other software. Currently mathematica and sage are supported.

9.1.25 ViewGraph (for IsObject, IsString)

▷ ViewGraph(*Graph, String*) (operation)

Returns: IsString.

This takes a graph and outputs code to view the graph in other software. Currently mathematica and sage are supported.

Chapter 10

Databases

10.1 Regular polyhedra

10.1.1 DegeneratePolyhedra

▷ `DegeneratePolyhedra(`*`sizerange`*`)` (function)

Returns: IsList

Gives all degenerate polyhedra (of type $\{2,q\}$ and $\{p,2\}$) with sizes in *`sizerange`*. Also accepts a single integer *maxsize* as input to indicate a sizerange of `[1..maxsize]`.

Example

```
gap> DegeneratePolyhedra(24);
[ AbstractRegularPolytope([ 2, 2 ]), AbstractRegularPolytope([ 2, 3 ]),
  AbstractRegularPolytope([ 3, 2 ]), AbstractRegularPolytope([ 2, 4 ]),
  AbstractRegularPolytope([ 4, 2 ]), AbstractRegularPolytope([ 2, 5 ]),
  AbstractRegularPolytope([ 5, 2 ]), AbstractRegularPolytope([ 2, 6 ]),
  AbstractRegularPolytope([ 6, 2 ]) ]
```

10.1.2 FlatRegularPolyhedra

▷ `FlatRegularPolyhedra(`*`sizerange`*`)` (function)

Returns: IsList

Gives all nondegenerate flat regular polyhedra with sizes in *`sizerange`*. Also accepts a single integer *maxsize* as input to indicate a sizerange of `[1..maxsize]`. Currently supports a maxsize of 4000 or less.

Example

```
gap> FlatRegularPolyhedra([10..24]);
[ AbstractRegularPolytope([ 2, 3 ]), AbstractRegularPolytope([ 3, 2 ]),
  AbstractRegularPolytope([ 2, 4 ]), AbstractRegularPolytope([ 4, 2 ]),
  AbstractRegularPolytope([ 2, 5 ]), AbstractRegularPolytope([ 5, 2 ]),
  AbstractRegularPolytope([ 4, 3 ], "r2 r1 r0 r1 = (r0 r1)^2 r1 (r1 r2)^1, r2 r1 r2 r1 r0 r1 = (r
 r1)^3 (r1 r2)^2"),
  ReflexibleManiplex([ 3, 4 ], "(r2*r1)^2*r1^2*r0*r1*r2*r1*r0,(r2*r1)^3*(r1*r0)^2*r1*r2*(r1*r0)^2
), AbstractRegularPolytope([ 2, 6 ]), AbstractRegularPolytope([ 6, 2 ]) ]
```

10.1.3 RegularToroidalPolyhedra44

▷ RegularToroidalPolyhedra44(*sizerange*) (function)

Returns: IsList

Gives all regular toroidal polyhedra of type $\{4,4\}$ with sizes in *sizerange*. Also accepts a single integer *maxsize* as input to indicate a sizerange of [1..maxsize].

Example

```
gap> RegularToroidalPolyhedra44([60..100]);
[ AbstractRegularPolytope([ 4, 4 ], "(r0 r1 r2)^4"),
  AbstractRegularPolytope([ 4, 4 ], "(r0 r1 r2 r1)^3") ]
```

10.1.4 RegularToroidalPolyhedra36

▷ RegularToroidalPolyhedra36(*sizerange*) (function)

Returns: IsList

Gives all regular toroidal polyhedra of type $\{3,6\}$ with sizes in *sizerange*. Also accepts a single integer *maxsize* as input to indicate a sizerange of [1..maxsize].

Example

```
gap> RegularToroidalPolyhedra36([100..150]);
[ AbstractRegularPolytope([ 3, 6 ], "(r0 r1 r2)^6"),
  AbstractRegularPolytope([ 3, 6 ], "(r0 r1 r2 r1 r2)^4") ]
```

10.1.5 SmallRegularPolyhedraFromFile

▷ SmallRegularPolyhedraFromFile(*sizerange*) (function)

Returns: IsList

Gives all regular polyhedra with sizes in *sizerange* flags that are stored separately in a file. These are polyhedra that are not part of one of several infinite families that are covered by the other generators. The return value of this function is unstable and may change as more infinite familes of polyhedra are identified and written as separate generators.

Example

```
gap> SmallRegularPolyhedraFromFile(64);
[ Simplex(3), AbstractRegularPolytope([ 3, 6 ], "(r2*r0*r1)^2*(r0*r2*r1)^2 "), CrossPolytope(3),
  AbstractRegularPolytope([ 6, 3 ], "(r0*r2*r1)^2*(r2*r0*r1)^2 "), Cube(3),
  AbstractRegularPolytope([ 5, 5 ], "r1*r2*r0*(r1*r0*r2)^2 "),
  AbstractRegularPolytope([ 3, 5 ], "(r2*r0*r1)^3*(r0*r2*r1)^2 "),
  AbstractRegularPolytope([ 5, 3 ], "(r0*r2*r1)^3*(r2*r0*r1)^2 ") ]
```

10.1.6 SmallRegularPolyhedra

▷ SmallRegularPolyhedra(*sizerange*) (function)

Returns: IsList

Gives all regular polyhedra with sizes in *sizerange* flags. Currently supports a maxsize of 4000 or less. You can also set options nondegenerate, nonflat, and nontoroidal.

Example

```
gap> L1 := SmallRegularPolyhedra(500);;
gap> L2 := SmallRegularPolyhedra(1000 : nondegenerate);;
gap> L3 := SmallRegularPolyhedra(2000 : nondegenerate, nonflat);;
gap> Length(SmallRegularPolyhedra(64));
53
```

10.1.7 SmallDegenerateRegular4Polytopes

▷ SmallDegenerateRegular4Polytopes(*sizerange*) (function)

Returns: IsList

Gives all degenerate regular 4-polytopes with sizes in *sizerange* flags. Currently supports a maxsize of 8000 or less.

Example

```
gap>  SmallDegenerateRegular4Polytopes([64]);
[ AbstractRegularPolytope([ 4, 2, 4 ]), AbstractRegularPolytope([ 2, 8, 2 ]),
  regular 4-polytope of type [ 4, 4, 2 ] with 64 flags,
  ReflexibleManiplex([ 2, 4, 4 ], "(r2*r1*r2*r3)^2,(r1*r2*r3*r2)^2") ]
```

10.1.8 SmallRegular4Polytopes

▷ SmallRegular4Polytopes(*sizerange*) (function)

Returns: IsList

Gives all regular 4-polytopes with sizes in *sizerange* flags. Currently supports a maxsize of 4000 or less.

Example

```
gap> SmallRegular4Polytopes([100]);
[ AbstractRegularPolytope([ 5, 2, 5 ]) ]
```

10.1.9 SmallChiralPolyhedra

▷ SmallChiralPolyhedra(*sizerange*) (function)

Returns: IsList

Gives all chiral polyhedra with sizes in *sizerange* flags. Currently supports a maxsize of 4000 or less.

Example

```
gap> SmallChiralPolyhedra(100);
[ AbstractRotaryPolytope([ 4, 4 ], "s1*s2^-2*s1^2*s2^-1,(s1^-1*s2^-1)^2"),
  AbstractRotaryPolytope([ 4, 4 ], "s2*s1^-1*s2*s1^2*s2^2*s1^-1,(s1^-1*s2^-1)^2"),
  AbstractRotaryPolytope([ 3, 6 ], "s2^-1*s1*s2^-2*s1^-1*s2*s1^-1*s2^-2,(s1^-1*s2^-1)^2"),
  AbstractRotaryPolytope([ 6, 3 ], "s1*s2^-1*s1^2*s2*s1^-1*s2*s1^2,(s2*s1)^2") ]
```

10.1.10 SmallChiral4Polytopes

▷ SmallChiral4Polytopes(*sizerange*) (function)

Returns: IsList

Gives all chiral 4-polytopes with sizes in *sizerange* flags. Currently supports a maxsize of 4000 or less.

Example

```
gap> SmallChiral4Polytopes([200..250]);
[ AbstractRotaryPolytope([ 3, 4, 4 ], "s3^-1*s2^-2*s1^-1*s3*s1,s2^-1*s3^-2*s2^2*s3,(s2^-1*s3^-1)^
  AbstractRotaryPolytope([ 4, 4, 3 ], "s1*s2^2*s3*s1^-1*s3^-1,s2*s1^2*s2^-2*s1^-1,(s2*s1)^2,(s3*s
  AbstractRotaryPolytope([ 4, 4, 4 ], "s2*s3^-2*s2^2*s3^-1,s3*s2*s1^-1*s3^2*s1,s3^-1*s2^-2*s1^-1*
```

10.1.11 SmallReflexible3Maniplexes

▷ SmallReflexible3Maniplexes(*sizerange*) (function)

Returns: IsList

Gives all regular 4-polytopes with sizes in *sizerange* flags. Currently supports a maxsize of 2000 or less. If the option nonpolytopal is set, only returns maniplexes that are not polyhedra.

10.1.12 SRP

▷ SRP(*minsize, maxsize[, func1, val1, func2, val2, ...]*) (function)

Returns: IsList

Returns a list of all regular polyhedra with at least *minsize* at at most *maxsize* flags. Optionally, you may include any number of pairs of functions and values, in which case this only returns those polyhedra such that the given functions have the given values. The name of this function is temporary and this function is here as a proof-of-concept.

Example

```
gap> SRP(1,200,SchlafliSymbol,[4,4]);
[ FlatOrientablyRegularPolyhedron(4,4,-1,1), AbstractRegularPolytope([ 4, 4 ], "(r0 r1 r2)^4"), A
gap> SRP(1,200,SchlafliSymbol,[4,4],IsFlat,false);
[ AbstractRegularPolytope([ 4, 4 ], "(r0 r1 r2)^4"), AbstractRegularPolytope([ 4, 4 ], "(r0 r1 r2
gap> SRP(1,32,p->SchlafliSymbol(p)[1], 4);
[ AbstractRegularPolytope([ 4, 2 ]), AbstractRegularPolytope([ 4, 3 ], "r2 r1 r0 r1 = (r0 r1)^2 r
```

10.2 System internal representations

10.2.1 DatabaseString (for IsManiplex)

▷ DatabaseString(*M*) (operation)

Returns: String

Given a maniplex *M*, returns a string representation of *M* suitable for saving in a database for later retrieval.

10.2.2 ManiplexFromDatabaseString (for IsString)

▷ ManiplexFromDatabaseString(*maniplexString*) (operation)

Returns: IsManiplex

Given a string *maniplexString*, representing a maniplex stored in a database, returns the maniplex that is represented.

Chapter 11

Stratified Operations

11.1 Computational tools

I should say something more here.

11.1.1 ChunkMultiply (for IsList,IsList)

▷ `ChunkMultiply(`*element1, element2*`)` (operation)

Returns: element

Elements are ordered pairs of the form [perm, list], where the elements of list are members of a group. Operation performed is consistent with that in defined in [PW18].

11.1.2 ChunkPower (for IsList,IsInt)

▷ `ChunkPower(`*element, integer*`)` (operation)

Returns: element

Given an element compatible with the ChunkMultiply operation, this function will compute the product of element with itself integer times.

11.1.3 ChunkGeneratedGroupElements (for IsList, IsGroup)

▷ `ChunkGeneratedGroupElements(`*list, group*`)` (operation)

Returns: newList

Given a list of generators compatible with the ChunkMultiply operation, this function will construct the associated list of group elements in a form suitable for taking ChunkMultiply and ChunkPower.

11.1.4 ChunkGeneratedGroup (for IsList, IsPermGroup)

▷ `ChunkGeneratedGroup(`*list, group*`)` (operation)

Returns: permGroup

Given a list of generators compatible with the ChunkMultiply operation, this function will construct a representation of the group as a permutation group. Note that generators are of the form [perm, list], and each list is a list of elements from group.

Example

```
gap> p:=Simplex(2); a:=AutomorphismGroup(p);
Pgon(3)
<fp group of size 6 on the generators [ r0, r1 ]>
gap> e:=One(a);; AssignGeneratorVariables(a);
gap> s0:=[(3,4),[r0,r0,e,e,r0,r0]];
[ (3,4), [ r0, r0, <identity ...>, <identity ...>, r0, r0 ] ]
gap> s1:=[(2,3)(4,5),[r1,e,e,e,e,r1]];
[ (2,3)(4,5), [ r1, <identity ...>, <identity ...>, <identity ...>, <identity ...>, r1 ] ]
gap> s2:=[(1,2)(5,6),[e,e,r1,r1,e,e]];
[ (1,2)(5,6), [ <identity ...>, <identity ...>, r1, r1, <identity ...>, <identity ...> ] ]
gap>  gens:=[s0,s1,s2];;
gap> ChunkMultiply(s0,s1);
[ (2,3,5,4), [ r0*r1, <identity ...>, r0, r0, <identity ...>, r0*r1 ] ]
gap> ChunkMultiply(s0,s0);
[ (), [ r0^2, r0^2, <identity ...>, <identity ...>, r0^2, r0^2 ] ]
gap> SetReducedMultiplication(r1);
gap> ChunkMultiply(s0,s0);
[ (), [ <identity ...>, <identity ...>, <identity ...>, <identity ...>, <identity ...>,  <identit
gap> ChunkGeneratedGroup(gens,a);
<permutation group with 3 generators>
gap> Size(last);
1296
```

11.1.5 FullyStratifiedGroup (for IsList, IsGroup)

▷ FullyStratifiedGroup(*list*, *group*) (operation)

Returns: IsPermGroup

Implements fully stratified operations on maniplexes from [CPW22]. Given *list* of generators compatible with the ChunkMultiply operation, *group* is the underlying group in the representation (usually the connection group of the base), this will calculate the connection group of the resulting maniplex acting on the implicit flags of the construction. Function assumes that *list* are the generators of the connection group of the resulting maniplex in the order $\langle r_0, r_1, \ldots, r_{d-1} \rangle$. It is possible that for some groups this function will behave poorly because GAP won't recognize equivalent representations of a group element. If so, try again with a permutation representation and let us know so we can modify the code to handle this problem better (didn't show up in our testing, but is a theoretical possibility).

Example

```
gap> p:=Simplex(2);; a:=AutomorphismGroup(p);
<fp group of size 6 on the generators [ r0, r1 ]>
gap> e:=One(a);
<identity ...>
gap> AssignGeneratorVariables(a);
#I  Assigned the global variables [ r0, r1 ]
gap> s0:=[(3,4),[r0,r0,e,e,r0,r0]];
[ (3,4), [ r0, r0, <identity ...>, <identity ...>, r0, r0 ] ]
gap> s1:=[(2,3)(4,5),[r1,e,e,e,e,r1]];
[ (2,3)(4,5), [ r1, <identity ...>, <identity ...>, <identity ...>, <identity ...>, r1 ] ]
gap> s2:=[(1,2)(5,6),[e,e,r1,r1,e,e]];
[ (1,2)(5,6), [ <identity ...>, <identity ...>, r1, r1, <identity ...>, <identity ...> ] ]
```

```
gap> gens:=[s0,s1,s2];
[ [ (3,4), [ r0, r0, <identity ...>, <identity ...>, r0, r0 ] ],
  [ (2,3)(4,5), [ r1, <identity ...>, <identity ...>, <identity ...>, <identity ...>, r1 ] ],
  [ (1,2)(5,6), [ <identity ...>, <identity ...>, r1, r1, <identity ...>, <identity ...> ] ] ]
gap> Maniplex(FullyStratifiedGroup(gens,a))=Prism(Simplex(2));
true
```

Chapter 12

Regular maps

12.1 Bicontactual regular maps

The names for the maps in this section are from S.E. Wilson's [Wil85].

12.1.1 Epsilonk (for IsInt)

▷ `Epsilonk(k)` (operation)

Returns: IsManiplex

Given an integer k, gives the map ε_k, which is $\{k,2\}_k$ when k is even, and $\{k,2\}_{2k}$ when k is odd.

Example

```
gap> Epsilonk(5);
AbstractRegularPolytope([ 5, 2 ])
gap> Epsilonk(6);
AbstractRegularPolytope([ 6, 2 ])
```

12.1.2 Deltak (for IsInt)

▷ `Deltak(k)` (operation)

Returns: IsManiplex

Given an integer k, gives the map δ_k, which is $\{2k,2\}/2$ when k is even, and $\{2k,2\}_k$ when k is odd.

Example

```
gap> Deltak(5);
AbstractRegularPolytope([ 10, 2 ], "(r0 r1)^5 r2")
gap> Deltak(6);
AbstractRegularPolytope([ 12, 2 ], "(r0 r1)^6 r2")
```

12.1.3 Mk (for IsInt)

▷ `Mk(k)` (operation)

Returns: IsManiplex

Given an integer k, gives the map M_k, which is $\{2k,2k\}_{1,0}$ when k is even, and $\{2k,k\}_2$ when k is odd.

Example

```
gap> Mk(5);Mk(6);
AbstractRegularPolytope([ 10, 5 ], "(r0 r1)^5 r0 = r2")
AbstractRegularPolytope([ 12, 12 ], "(r0 r1)^6 r0 = r2")
```

12.1.4 MkPrime (for IsInt)

▷ `MkPrime(k)` (operation)

Returns: IsManiplex

Given an integer k, gives the map M'_k, which is $\{k,k\}_2$ when k is even, and $\{k,2k\}_2$ when k is odd. `MkPrime(k,i)` gives the map $M'_{k,i}$.

Example

```
gap> MkPrime(5);MkPrime(6);
ReflexibleManiplex([ 5, 10 ], "(r2*r1*(r0 r2))^5,z1^2")
ReflexibleManiplex([ 6, 6 ], "(r2*r1*(r0 r2))^6,z1^2")
```

12.1.5 Bk2l (for IsInt,IsInt)

▷ `Bk2l(k, l)` (operation)

Returns: IsManiplex

Given integers k,l, gives the map $B(k,2l)$.

Example

```
gap> Bk2l(4,5);
3-maniplex with 80 flags
```

12.1.6 Bk2lStar (for IsInt,IsInt)

▷ `Bk2lStar(k, l)` (operation)

Returns: IsManiplex

Given integers k,l, gives the map $B^*(k,2l)$.

Example

```
gap> Bk2lStar(5,7);
3-maniplex with 140 flags
```

12.2 Operators on reflexible maps

12.2.1 Opp (for IsManiplex)

▷ `Opp(map)` (operation)

Returns: IsManiplex

Forms the opposite map of the maniplex map.

Example

```
gap> Opp(Bk2lStar(5,7));
Petrial(Dual(Petrial(3-maniplex with 140 flags)))
```

12.2.2 Hole (for IsManiplex,IsInt)

▷ `Hole(map, j)` (operation)

Returns: IsManiplex

Given *map* and integer j, will form the map $H_j(map)$. Note that if the action of $[r_0,(r_1r_2)^{j-1}r_1,r_2]$ on the flags forms multiple orbits, then the resulting map will be on just one of those orbits.

Example

```
gap> Hole(Bk2lStar(5,7),2);
3-maniplex with 140 flags
```

Chapter 13

Utility functions

13.1 Utility functions

13.1.1 InfoRamp

▷ `InfoRamp` (info class)

The InfoClass for the Ramp package.

13.1.2 AbstractPolytope

▷ `AbstractPolytope(`*`args`*`)` (function)

Calls `Maniplex(args)` and marks the output as polytopal.

13.1.3 AbstractRegularPolytope

▷ `AbstractRegularPolytope(`*`args`*`)` (function)

Calls `ReflexibleManiplex(args)` and marks the output as polytopal. Also available as `ARP(args)`.

13.1.4 AbstractRotaryPolytope

▷ `AbstractRotaryPolytope(`*`args`*`)` (function)

Calls `RotaryManiplex(args)` and marks the output as polytopal.

13.1.5 TranslatePerm

▷ `TranslatePerm(`*`perm, k`*`)` (function)

Returns a new permutation obtained from *perm* by adding k to each moved point.

13.1.6 MultPerm

▷ MultPerm(*perm, multiplier, offset*) (function)

Multiplies together perm, TranslatePerm(perm, offset), TranslatePerm(perm, offset*2), ..., with *multiplier* terms, and returns the result.

13.1.7 PermFromRange

▷ PermFromRange(*perm1, perm2*) (function)

Returns: Permutation

This attempts to construct a permutation that we would write as perm1 ... perm2. Probably it is clearest to look at some examples:

Example

```
gap> PermFromRange((1,2), (9,10));
(1,2)(3,4)(5,6)(7,8)(9,10)
gap> PermFromRange((1,3), (13,15));
(1,3)(4,6)(7,9)(10,12)(13,15)
gap> PermFromRange((2,3,4), (8,9,10));
(2,3,4)(5,6,7)(8,9,10)
```

13.1.8 ParseStringCRels

▷ ParseStringCRels(*rels, g*) (function)

Returns: a list of relators

This helper function is used in several maniplex constructors. Given a string *rels* that represents relations in an sggi, and an sggi g, returns a list of elements in the free group of g represented by *rels*. These can then be used to form a quotient of g.

Example

```
gap> g := AutomorphismGroup(CubicTiling(2));;
gap> rels := "(r0 r1 r2 r1)^6";;
gap> newrels := ParseStringCRels(rels, g);
[ (r0*r1*r2*r1)^6 ]
gap> newrels[1] in FreeGroupOfFpGroup(g);
true
gap> g2 := FactorGroupFpGroupByRels(g, newrels);
<fp group on the generators [ r0, r1, r2 ]>
```

For convenience, you may use z1, z2, etc and h1, h2, etc in relations, where zj means r0 (r1 r2)^j (the "j-zigzag" word) and hj means r0 (r1 r2)^j-1 r1 (the "j-hole" word).

13.1.9 ParseRotGpRels

▷ ParseRotGpRels(*rels, g*) (function)

This helper function is used in several maniplex constructors. It is analogous to ParseStringCRels, but for rotation groups instead.

13.1.10 AddOrAppend

▷ AddOrAppend(*L, x*) (function)

Given a list *L* and an object *x*, this calls Append(L, x) if x is a list; otherwise it calls Add(L, x). Note that since strings are internally represented as lists, AddOrAppend(L, "foo") will append the characters 'f', 'o', 'o'.

Example

```
gap> L := [1, 2, 3];;
gap> AddOrAppend(L, 4);
gap> L;
[1, 2, 3, 4]
gap> AddOrAppend(L, [5, 6]);
gap> L;
[1, 2, 3, 4, 5, 6];
```

13.1.11 WrappedPosetOperation

▷ WrappedPosetOperation(*posetOp*) (function)

Given a poset operation, creates a bare-bones maniplex operation that delegates to the poset operation.

Example

```
gap> myjoin := WrappedPosetOperation(JoinProduct);
function( arg... ) ... end
gap> M := myjoin(Pgon(4), Vertex());
3-maniplex
gap> M = Pyramid(4);
true
```

Usually, you will want to eventually create a fuller-featured wrapper of the poset operation – one that can infer more information from its arguments. But this method is a good way to quickly test whether a poset operation works on maniplexes the way one expects.

References

[CM17] Gabe Cunningham and Mark Mixer. Internal and external duality in abstract polytopes. *Contrib. Discrete Math.*, 12(2):187–214, 2017. 13, 61

[CPW22] Gabe Cunningham, Daniel Pellicer, and Gordon Ian Williams. Stratified operations on maniplexes. *Algebr. Comb.*, 2022. 78

[Cun21] Gabe Cunningham. Flat extensions of abstract polytopes. *Art Discrete Appl. Math.*, 4(3):Paper No. 3.06, 14, 2021. 60

[GH18] Ian Gleason and Isabel Hubard. Products of abstract polytopes. *Journal of Combinatorial Theory, Series A*, 157:287–320, jul 2018. 56

[HW10] Michael I. Hartley and Gordon I. Williams. Representing the sporadic archimedean polyhedra as abstract polytopes. *Discrete Mathematics*, 310(12):1835–1844, jun 2010. 15

[MPW12] Barry Monson, Daniel Pellicer, and Gordon Williams. The tomotope. *Ars Mathematica Contemporanea*, 5(2):355–370, jun 2012. 14

[MPW14] B. Monson, Daniel Pellicer, and Gordon Williams. Mixing and monodromy of abstract polytopes. *Transactions of the American Mathematical Society*, 366(5):2651–2681, nov 2014. 42

[MS02] Peter McMullen and Egon Schulte. *Abstract Regular Polytopes*. Cambridge University Press, dec 2002. 42, 46

[Pel18] Daniel Pellicer. Cleaved abstract polytopes. *Combinatorica*, 38(3):709–737, mar 2018. 46

[PW18] Daniel Pellicer and Gordon Ian Williams. Pyramids over regular 3-tori. *SIAM Journal on Discrete Mathematics*, 32(1):249–265, jan 2018. 77

[Wil85] Stephen Wilson. Bicontactual regular maps. *Pacific Journal of Mathematics*, 120(2):437–451, dec 1985. 80

[Wil12] Steve Wilson. Maniplexes: Part 1: Maps, polytopes, symmetry and operators. *Symmetry*, 4(2):265–275, apr 2012. 42

Index