RAMP

The Research Assistant for Maniplexes and Polytopes

0.7.01

1 July 2022

Gabe Cunningham

Mark Mixer

Gordon Williams

Gabe Cunningham
Email: gabriel.cunningham@gmail.com
Homepage: http://www.gabrielcunningham.com

Mark Mixer
Email: mixerm@wit.edu

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

Copyright

© 1997-2022 by Gabe Cunningham, Mark Mixer, and Gordon Williams

RAMP package is free software; you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation; either version 2 of the License, or (at your option) any later version.

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

Installation

1.1 Basics

Some quick notes on installation:

- RAMP is confirmed to work with version 4.11.1 of GAP, but is known not to work with some earlier versions.
- Copy the RAMP folder and its contents to your GAP `/pkg` folder.
  - If using the GAP.app on macOS, this should be your user `Library/Preferences/GAP/pkg` folder. Probably the easiest way to do this if you have received RAMP as a `.zip` file is to copy the file into this location, and then unpack it. After that, you can delete the `.zip` file.

Chapter 2

Using RAMP

2.1 Assumptions

There are a few assumptions that many methods make.

1. The connection group of a maniplex with N flags is often assumed to act on [1..N]. This is gradually being rewritten to allow any set of integers as flags, but use caution when working with such connection groups.

2. When working with a connection group $\langle r_0, \ldots, r_{n-1} \rangle$, some methods may have strange behavior if any r_i or $r_i r_j$ has any fixed points. Indeed, Sggis of that type define pre-maniplexes rather than maniplexes. Eventually, the methods that build maniplexes will verify that no r_i or $r_i r_j$ has fixed points.

2.2 Extending RAMP

Suppose you want to add a new operation on maniplexes to RAMP. We will see how to accomplish that with a hypothetical example. Let's pretend that there's a mathematical operation on maniplexes called "Stretch".

Our first step will be to create two new files in the lib/ directory: stretch.gd and stretch.gi. The first file is for the declaration of the new operation, and the second is for the implementation.

In stretch.gd, we want to add a line that declares the new operation, something like this:

Example
```
DeclareOperation("Stretch", IsManiplex);
```

Now we will write the implementation in stretch.gi:

Example
```
InstallMethod(Stretch,
                [IsManiplex],
                function(M)
                ...actual code goes here...
                end);
```

Finally, we need to make sure that these new files are read when RAMP is loaded up. Open up init.g (in the root RAMP directory) and add the line

Example
```
ReadPackage( "ramp", "lib/stretch.gd" );
```

(We recommend that you put that line in alphabetical order with the rest.) Similarly, open up read.g and add the line

Example
```
ReadPackage( "ramp", "lib/stretch.gi" );
```

Now your code is available in your copy of RAMP!

Here are two more things you should do. First, test your code. Create a new file called stretch.tst in the tst directory of RAMP. The format of the tests is that you first write a line that starts with "gap> " and continues with some input, as if you actually typed it in to the GAP prompt. Then, on the following line, put the expected output.

To run your tests, run the following command in RAMP:

Example
```
gap> TestRamp("stretch.tst");
```

If any tests fail (that is, if the output from GAP does not match the expected output from your test file), then GAP will alert you to the discrepancies. Otherwise, when the tests are complete, there will be no output and you will just see the gap> prompt again.

You can also call TEST_RAMP() to run all of the tests in the /tst directory.

Finally, you should document your operation! Take a look at one of the .gd files included with RAMP to see what you should include. To actually build the documentation, you will need the package AutoDoc. For example, the following will rebuild the documentation:

Example
```
gap> LoadPackage("AutoDoc");
gap> AutoDoc("ramp", rec( scaffold := true, autodoc := true));
```

To see your updated documentation, you can either navigate to the html file in the doc/ directory, or you can quit GAP and restart it, and then your documentation will be available in the inline help. If you have LaTeX set up properly, then it will also build a pdf manual.

Chapter 3

Group Constructors

3.1 Ggis

3.1.1 UniversalGgi

▷ `UniversalGgi(n)` (operation)
▷ `UniversalGgi(cox)` (operation)

Returns: `IsFpGroup`

In the first form, returns the universal Coxeter Group of rank n. In the second form, returns the Coxeter Group with the given Coxeter diagram. The diagram is given as a list with the order of r0 r1, r0 r2, ..., r0 r_{n-1}, r_1 r_2, etc.

Example
```
gap> g := UniversalGgi(3);
<fp group of size infinity on the generators [ r0, r1, r2 ]>
gap> RelatorsOfFpGroup(g);
[ r0^2, r1^2, r2^2 ]
gap> q := UniversalGgi([3,3,3]);
<fp group on the generators [ r0, r1, r2 ]>
gap> RelatorsOfFpGroup(q);
[ r0^2, r1^2, r2^2, (r0*r1)^3, (r0*r2)^3, (r1*r2)^3 ]
```

3.1.2 Ggi

▷ `Ggi(cox[, relations])` (operation)
▷ `Ggi(cox, words, orders)` (operation)

Returns: `IsFpGroup`

Returns the ggi defined by the given Coxeter diagram 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 ggi with the given Coxeter diagram. This method automatically calls `InterpolatedString` on the relations, so you may use $variable in the relations, and it will be replaced with the value of `variable` (but for global variables only).

Example
```
gap> g := Ggi([3,3,3], "(r0 r1 r2 r1)^3");;
gap> Size(g);
54
gap> n := 5;;
```

```
gap> Size(Ggi([3,3,3], "(r0 r1 r2 r1)^$n"));
150
```

The second form takes the Coxeter diagram *cox*, a list of *words* in the generators r0 etc, and a list of *orders*. It returns the ggi that is the quotient of the universal ggi with that Coxeter diagram by the relations obtained by setting each *word[i]* to have order *order[i]*. This is primarily useful for quickly constructing a family of related Ggis.

Example
```
gap> L := List([1..5], k -> Ggi([3,3,3], ["r0 r1 r2 r1"], [k]));;
gap> List(L, Size);
[ 6, 24, 54, 96, 150 ]
```

3.1.3 CyclicGgi (for IsList, IsList)

▷ `CyclicGgi(cox[, relations])` (operation)

Returns: `IsFpGroup`

Returns the ggi with a cyclic diagram defined by the given orders, and subject to the given relations. *cox* gives the orders of r0 r1, r1 r2, ..., r_{n-1} r_0.

Example
```
gap> g := CyclicGgi([3,4,5,6]);
<fp group on the generators [ r0, r1, r2, r3 ]>
gap> RelatorsOfFpGroup(g);
[ r0^2, r1^2, r2^2, r3^2, (r0*r1)^3, (r0*r2)^2, (r0*r3)^6, (r1*r2)^4, (r1*r3)^2, (r2*r3)^5 ]
gap> g2 := CyclicGgi([3,4,3,4], "(r0 r1 r2 r3)^4");
<fp group on the generators [ r0, r1, r2, r3 ]>
gap> Size(g2);
1440
```

3.1.4 GgiElement (for IsGroup, IsString)

▷ `GgiElement(g, str)` (operation)

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

This method automatically calls `InterpolatedString` on the relations, so you may use $variable in the relations, and it will be replaced with the value of `variable` (but for global variables only).

Example
```
gap> gap> g := Group((1,2), (2,3), (3,4), (1,4));;
gap> GgiElement(g, "r0 r3");
(1,2,4)
```

3.1.5 SimplifiedGgiElement (for IsGroup, IsString)

▷ `SimplifiedGgiElement(g, str)` (operation)

Returns: the element of *g* with underlying word *str*, in a reduced form.

This acts like GgiElement, except that the word is in reduced form. Note that this is accomplished by calling SetReducedMultiplication on g. As a result, further computations with g may be substantially slower. This method automatically calls `InterpolatedString` on the relations, so you may use $variable in the relations, and it will be replaced with the value of `variable` (but for global variables only).

Example
```
gap> g := Ggi([3,3,3], "(r0 r1 r2 r1)^4");;
gap> SimplifiedGgiElement(g, "(r0 r1)^4");
r0*r1
```

3.2 Sggis

3.2.1 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.

Example
```
gap> g:=UniversalSggi(3);
<fp group of size infinity on the generators [ r0, r1, r2 ]>
gap> q:=UniversalSggi([3,4]);
<fp group of size 48 on the generators [ r0, r1, r2 ]>
gap> IsQuotient(g,q);
true
```

3.2.2 Sggi

▷ `Sggi(symbol[, relations])` (operation)
▷ `Sggi(sym, words, orders)` (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. This method automatically calls `InterpolatedString` on the relations, so you may use $variable in the relations, and it will be replaced with the value of `variable` (but for global variables only).

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
gap> n := 3;;
gap> Size(Sggi([4,4], "(r0 r1 r2 r1)^$n"));
72
```

The second form takes the Schlafli Symbol *sym*, a list of *words* in the generators r0 etc, and a list of *orders*. It returns the Sggi that is the quotient of the universal Sggi with that Schlafli Symbol by the relations obtained by setting each *word[i]* to have order *order[i]*. This is primarily useful for quickly constructing a family of related Sggis.

Example
```
gap> L := List([1..5], k -> Sggi([4,4], ["r0 r1 r2"], [2*k]));;
gap> List(L, Size);
[ 16, 64, 144, 256, 400 ]
```

3.2.3 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.

Example
```
gap> IsGgi(SymmetricGroup(4));
false
gap> IsGgi(Group([(1,2),(2,3)]));
true
```

3.2.4 IsStringy (for IsGroup)

▷ `IsStringy(g)` (property)

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

Example
```
gap> IsStringy(Group((1,2),(2,3),(3,4)));
true
gap> IsStringy(Group((1,2),(3,4),(2,3)));
false
```

3.2.5 IsSggi (for IsGroup)

▷ `IsSggi(g)` (property)

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

Example
```
gap> IsSggi(SymmetricGroup(4));
false
gap> IsSggi(Group((1,2),(3,4),(2,3)));
false
gap> IsSggi(Group((1,2),(2,3),(3,4)));
true
```

3.2.6 IsFixedPointFreeSggi (for IsGroup)

▷ `IsFixedPointFreeSggi(g)` (property)

Returns: whether g is a string group generated by involutions such that no generator and no product of two generators has any fixed points. A premaniplex M is a maniplex if and only if IsFixedPointFreeSggi(ConnectionGroup(M)). Equivalent to `IsGgi(g) and IsStringy(g)`.

Example
```
gap> IsFixedPointFreeSggi(Group((1,2)(3,4), (1,3)(2,4) ,(1,4)(2,3)));
true
gap> IsFixedPointFreeSggi(Group((1,2)(3,4), (1,2)(3,4), (1,4)(2,3)));
false
```

3.2.7 IsStringRotationGroup (for IsGroup)

▷ `IsStringRotationGroup(g)` (property)

Returns: Whether *g* is a string rotation group, i.e. the even word subgroup of an Sggi. This means that the product of adjacent generators should be an involution.

Example
```
gap> IsStringRotationGroup(Group((1,2)(3,4), (2,3,4)));
false
gap> IsStringRotationGroup(Group((1,3,2), (2,4,3)));
true
```

3.2.8 IsStringC (for IsGroup)

▷ `IsStringC(G)` (property)

Returns: Whether `G` is a string C group. Currently only works for finite groups.

Example
```
gap> IsStringC(Sggi([4,4], "r0 r1 r2"));
false
gap> IsStringC(Sggi([4,4], "(r0 r1 r2)^4"));
true
```

3.2.9 IsStringCPlus (for IsGroup)

▷ `IsStringCPlus(G)` (property)

Returns: Whether `G` is a string C+ group. Currently only works for finite groups.

Example
```
gap> IsStringCPlus(Group((1,2)(3,4), (2,3,4)));
false
gap> IsStringCPlus(Group((1,3,2), (2,4,3)));
true
gap> IsStringCPlus(RotationGroup(ToroidalMap44([1,0])));
false
```

3.2.10 SggiElement (for IsGroup, IsString)

▷ `SggiElement(g, str)` (operation)

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

This method automatically calls `InterpolatedString` on the relations, so you may use $variable in the relations, and it will be replaced with the value of `variable` (but for global variables only).

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

```
(1,3,2)
gap> n := 2;;
gap> SggiElement(g, "(r0 r1)^$n");
(1,2,3)
```

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

3.2.11 SimplifiedSggiElement (for IsGroup, IsString)

▷ `SimplifiedSggiElement(g, str)` (operation)

Returns: the element of *g* with underlying word *str*, in a reduced form.

This acts like SggiElement, except that the word is in reduced form. Note that this is accomplished by calling SetReducedMultiplication on g. As a result, further computations with g may be substantially slower. This method automatically calls `InterpolatedString` on the relations, so you may use $variable in the relations, and it will be replaced with the value of `variable` (but for global variables only). For convenience, you can also use a reflexible maniplex M in place of g, in which case `AutomorphismGroup(M)` is used for g.

Example
```
gap> g := AutomorphismGroup(Cube(3));;
gap> SimplifiedSggiElement(g, "(r0 r1)^5");
r0*r1
```

3.2.12 IsRelationOfReflexibleManiplex (for IsManiplex, IsString)

▷ `IsRelationOfReflexibleManiplex(M, rel)` (operation)

Returns: IsBool

Determines whether the relation given by the string *rel* holds in `AutomorphismGroup(M)`. This method automatically calls `InterpolatedString` on the relations, so you may use $variable in the relations, and it will be replaced with the value of `variable` (but for global variables only).

Example
```
gap> M := ReflexibleManiplex([8,6],"(r0 r1)^4 (r1 r2)^3");;
gap> IsRelationOfReflexibleManiplex(M, "(r0 r1 r2)^3");
false
gap> IsRelationOfReflexibleManiplex(M, "(r0 r1 r2)^12");
true
```

3.2.13 RotGpElement (for IsGroup, IsString)

▷ `RotGpElement(g, str)` (operation)

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

This method automatically calls `InterpolatedString` on the relations, so you may use $variable in the relations, and it will be replaced with the value of `variable` (but for global variables only).

Example
```
gap> Order(RotGpElement(Cube(3), "s1"));
4
gap> Order(RotGpElement(ToroidalMap44([1,2]), "s2^-1 s1"));
5
```

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

3.2.14 SimplifiedRotGpElement (for IsGroup, IsString)

▷ `SimplifiedRotGpElement(g, str)` (operation)

Returns: the element of the rotation group *g* with underlying word *str*, in a reduced form.

This acts like RotGpElement, except that the word is in reduced form. Note that this is accomplished by calling SetReducedMultiplication on g. As a result, further computations with g may be substantially slower. This method automatically calls `InterpolatedString` on the relations, so you may use $variable in the relations, and it will be replaced with the value of `variable` (but for global variables only). For convenience, you can also use a rotary maniplex M in place of g, in which case `RotationGroup(M)` is used for g.

Example
```
gap> SimplifiedRotGpElement(ToroidalMap44([1,2]), "s1^6");
s1^2
```

3.2.15 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.

3.2.16 IsCConnected (for IsManiplex)

▷ `IsCConnected(m)` (property)

Returns: IsBool

Determines whether a given maniplex is C-connected (i.e., is the connection group a string C-group).

Example
```
gap> IsCConnected(ToroidalMap44([1,0]));
false
gap> IsCConnected(Prism(ToroidalMap44([1,0])));
true
```

3.2.17 SectionSubgroup (for IsGroup, IsList)

▷ SectionSubgroup(*g*, *I*) (operation)

Returns: IsSggi

Given an Sggi g, returns the subgroup generated by those generators with indices in *I*.

Example
```
gap> g := AutomorphismGroup(Cube(5));;
gap> SectionSubgroup(g, [0, 2, 3]);
Group([ r0, r2, r3 ])
gap> Size(last);
12
```

3.2.18 VertexFigureSubgroup (for IsGroup)

▷ VertexFigureSubgroup(*g*) (operation)

Returns: IsSggi

Given an Sggi g, returns the vertex-figure subgroup; that is, the subgroup generated by all generators except for the first one.

Example
```
gap> VertexFigureSubgroup(AutomorphismGroup(Cube(3)));
Group([ r1, r2 ])
gap> Size(last);
6
```

3.2.19 FacetSubgroup (for IsGroup)

▷ FacetSubgroup(*g*) (operation)

Returns: IsSggi

Given an Sggi g, returns the facet subgroup; that is, the subgroup generated by all generators except for the last one.

Example
```
gap> FacetSubgroup(AutomorphismGroup(Cube(3)));
Group([ r0, r1 ])
gap> Size(last);
8
```

3.3 String rotation groups

3.3.1 UniversalRotationGroup (for IsInt)

▷ UniversalRotationGroup(*n*) (operation)

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

Example
```
gap> UniversalRotationGroup(3);
<fp group of size infinity on the generators [ s1, s2 ]>
```

3.3.2 UniversalRotationGroup (for IsList)

▷ `UniversalRotationGroup(sym)` (operation)

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

Example

```
gap> UniversalRotationGroup([4,4]);
<fp group of size infinity on the generators [ s1, s2 ]>
gap> UniversalRotationGroup([3,3,3]);
<fp group of size 60 on the generators [ s1, s2, s3 ]>
```

Chapter 4

Maniplex Constructors

4.1 Maniplexes

4.1.1 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*. This function first checks whether g is an Sggi. Use `ManiplexNC` to bypass that check.

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
```

4.1.2 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
```

4.1.3 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.

4.1.4 Maniplex (for IsPoset)

▷ `Maniplex(P)` (operation)

Returns: `IsManiplex`

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

4.1.5 Maniplex (for IsEdgeLabeledGraph)

▷ `Maniplex(P)` (operation)

Returns: `IsManiplex`

Constructs the maniplex from its flag graph *F*. This assumes that F actually defines a maniplex.

4.1.6 Premaniplex (for IsGroup)

▷ `Premaniplex(g)` (operation)

Returns: `IsPremaniplex`.

Given a group *g* we return the premaniplex with connection group *g*. This function first checks whether *group* is an Sggi. Use `PremaniplexNC` to bypass that check.

Here we build a premaniplex with 3 flags.

Example
```
gap> g:=Group((1,2),(2,3),(1,2));;
gap> Premaniplex(g);
Premaniplex of rank 3 with 3 flags
```

4.1.7 Premaniplex (for IsEdgeLabeledGraph)

▷ `Premaniplex(G)` (operation)

Returns: `IsPremaniplex`.

Given an edge labeled graph *G* we return the premaniplex with that graph. Note: We will assume (but not check) that the graph is a premaniplex, that is to say, we are assumging each vertex is incident with one edge of each label.

Here we have a premaniplex with 2 flags.

Example
```
gap> gap> L:=[[[1,2],0], [[1,2],1], [[1],2], [[2],2]];;
gap> F:=EdgeLabeledGraphFromLabeledEdges(L);;
gap> Premaniplex(F);
Premaniplex of rank 3 with 2 flags
```

4.2 Reflexible Maniplexes

4.2.1 ReflexibleManiplex

▷ `ReflexibleManiplex(sym[, relations])` (operation)
▷ `ReflexibleManiplex(sym, words, orders)` (operation)
▷ `ReflexibleManiplex(name)` (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. Note, however, that this function does not check whether the generators are all nontrivial and pairwise distinct, and so the output could be a pre-maniplex that is incorrectly labeled as a maniplex. For most purposes, this is relatively harmless, since most functions treat maniplexes and pre-maniplexes in roughly the same way.

For more information on relators, see the documentation on ParseGgiRels. 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. This method automatically calls `InterpolatedString` on the relations, so you may use $variable in the relations, and it will be replaced with the value of `variable` (but for global variables only).

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");;
gap> n := 3;;
gap> Size(ReflexibleManiplex([4,4], "(r0 r1 r2 r1)^$n"));
72
```

The third form takes the Schlafli Symbol *sym*, a list of *words* in the generators r0 etc, and a list of *orders*. It returns the reflexible maniplex that is the quotient of the universal maniplex with that Schlafli Symbol by the relations obtained by setting each *word[i]* to have order *order[i]*. This is primarily useful for quickly constructing a family of related maniplexes.

Example
```
gap> L := List([1..5], k -> ReflexibleManiplex([4,4], ["r0 r1 r2 r1"], [k]));;
gap> List(L, IsPolytopal);
[ false, true, true, true, true ]
```

The fourth form accepts common names for reflexible 3-maniplexes that specify the lengths of holes and zigzags. That is, "{p, q | h2, \ldots, hk\}_z1, ..., zL" is the quotient of $\{p,q\}$ by the relations that make the 2-holes have length h2, ..., and the 1-zigzags have length z1, etc.

Example
```
gap> ReflexibleManiplex("{4,4 | 6}") = ToroidalMap44([6,0]);
true
gap> ReflexibleManiplex("{4,4}_4") = ToroidalMap44([2,2]);
true
gap> M := ReflexibleManiplex("{6,6 | 6,2}_4");;
gap> HoleLength(M,2);
6
gap> HoleLength(M,3);
2
gap> ZigzagLength(M,1);
4
```

In the second and third forms, 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.

The abbreviations `RefMan` and `RefManNC` are also available for all of these usages.

4.2.2 ReflexiblePremaniplex (for IsGroup)

▷ `ReflexiblePremaniplex(`*g*`)` (operation)

Returns: `IsPremaniplex`

In the first form, we are given an Sggi g and we return the reflexible premaniplex 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 := ReflexiblePremaniplex(g);
gap> M = Simplex(3);
true
```

This function first checks whether g is an Sggi. Use `ReflexiblePremaniplexNC` to bypass that check.

4.3 Rotary Maniplexes

4.3.1 RotaryManiplex

▷ `RotaryManiplex(`*g*`)` (operation)
▷ `RotaryManiplex(`*sym*`)` (operation)
▷ `RotaryManiplex(`*sym, relations*`)` (operation)
▷ `RotaryManiplex(`*sym, words, orders*`)` (operation)

In the first form, 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). This function first checks whether g is a StringRotationGroup. Use `RotaryManiplexNC` to bypass that check.

Example
```
gap> M := RotaryManiplex(UniversalRotationGroup([3,3]));;
gap> M = Simplex(3);
true
```

The second form returns the universal rotary maniplex (in fact, regular polytope) with Schlafli symbol *sym*.

Example

```
gap> M := RotaryManiplex([4,3]);;
gap> M = Cube(3);
true
```

The third form 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. This method automatically calls `InterpolatedString` on the relations, so you may use $variable in the relations, and it will be replaced with the value of `variable` (but for global variables only).

Example

```
gap> M := RotaryManiplex([4,4], "(s2^-1 s1)^6");;
gap> M = ToroidalMap44([6,0]);
true
```

The fourth form takes the Schlafli Symbol *sym*, a list of *words* in the generators r0 etc, and a list of *orders*. It returns the rotary maniplex that is the quotient of the universal maniplex with that Schlalfi Symbol by the relations obtained by setting each *word[i]* to have order *order[i]*. This is primarily useful for quickly constructing a family of related maniplexes.

Example

```
gap> L := List([1..5], k -> RotaryManiplex([4,4], ["s1 s2^-1"], [k]));;
gap> List(L, IsPolytopal);
[ false, true, true, true, true ]
```

In the last two forms, 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.

Example

```
gap> M := RotaryManiplex([6,6], "(s1^2 s2^2)^8");;
gap> SchlafliSymbol(M);
#I  Coset table calculation failed -- trying with bigger table limit
... eventually give up with CTRL-C
gap> M := RotaryManiplex([6,6], "(s1^2 s2^2)^8" : set_schlafli);;
gap> SchlafliSymbol(M);
[6, 6]
```

4.3.2 EnantiomorphicForm (for IsManiplex)

▷ `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.

Example

```
gap> M := ToroidalMap44([1,2]);;
gap> g := AutomorphismGroup(M);
<fp group of size 20 on the generators [ s1, s2 ]>
```

```
gap> RelatorsOfFpGroup(g);
[ (s1*s2)^2, s1^4, s2^4, s2^-1*s1*(s2*s1^-1)^2 ]
gap> h := AutomorphismGroup(EnantiomorphicForm(M));
<fp group of size 20 on the generators [ s1, s2 ]>
gap> RelatorsOfFpGroup(h);
[ (s1*s2)^2, s1^4, s2^4, s2^-1*s1^-1*s2*s1^3*s2*s1 ]
```

4.4 Subclasses of maniplex

4.4.1 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.

4.4.2 IsPolytopal (for IsPremaniplex)

▷ `IsPolytopal(arg)` (property)

Returns: `true` or `false`

4.4.3 SatisfiesPathIntersectionProperty (for IsManiplex)

▷ `SatisfiesPathIntersectionProperty(M)` (property)

Returns: IsBool

Tests for the weak path intersection property in a maniplex. Definitions and description available in [GVH18].

4.4.4 IsFaithful (for IsManiplex)

▷ `IsFaithful(m)` (operation)

Returns whether the maniplex *m* is faithful, as defined in "Polytopality of Maniplexes"; i.e., whether for each flag the intersection of all the i-faces containing that flag is just the flag itself.

Example

```
gap> IsFaithful(Cube(3));
true
gap> IsFaithful(ToroidalMap44([1,0]));
false
```

4.4.5 IsRegularPolytope (for IsManiplex)

▷ `IsRegularPolytope(maniplex)` (attribute)

Returns whether a maniplex is a regular polytope.

Example

```
gap> p:=24Cell();
24Cell
gap> IsRegularPolytope(p);
```

```
true
gap> q:=CartesianProduct(Simplex(2),Cube(2));
CartesianProduct(Pgon(3), Pgon(4))
gap> IsRegularPolytope(q);
false
```

Chapter 5

Families of Polytopes

5.1 Classical polytopes

5.1.1 Vertex

▷ `Vertex()` (operation)

Returns: IsPolytope

Returns the universal 0-polytope.

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

5.1.2 Edge

▷ `Edge()` (operation)

Returns: IsPolytope

Returns the universal 1-polytope.

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

5.1.3 Pgon (for IsInt)

▷ `Pgon(p)` (operation)

Returns: IsPolytope

Returns the p-gon.

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

5.1.4 Cube (for IsInt)

▷ `Cube(n)` (operation)

Returns: IsPolytope

Returns the n-cube.

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

5.1.5 HemiCube (for IsInt)

▷ `HemiCube(n)` (operation)

Returns: IsPolytope

Returns the n-hemi-cube.

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

5.1.6 CrossPolytope (for IsInt)

▷ `CrossPolytope(n)` (operation)

Returns: IsPolytope

Returns the n-cross-polytope.

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

5.1.7 Octahedron

▷ `Octahedron()` (operation)

Returns: IsPolytope

Returns the octahedron (3-cross-polytope).

Example
```
gap> Octahedron() = CrossPolytope(3)
true
```

5.1.8 HemiCrossPolytope (for IsInt)

▷ `HemiCrossPolytope(n)` (operation)

Returns: IsPolytope

Returns the n-hemi-cross-polytope.

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

5.1.9 Simplex (for IsInt)

▷ `Simplex(n)` (operation)

Returns: IsPolytope

Returns the n-simplex.

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

5.1.10 Tetrahedron

▷ `Tetrahedron()` (operation)

Returns: IsPolytope

Returns the tetrahedron (3-simplex).

Example
```
gap> Tetrahedron() = Simplex(3)
true
```

5.1.11 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 ]
```

5.1.12 Dodecahedron

▷ `Dodecahedron()` (operation)

Returns: IsPolytope

Returns the dodecahedron, `{5, 3}`.

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

5.1.13 HemiDodecahedron

▷ `HemiDodecahedron()` (operation)

Returns: IsPolytope

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

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

5.1.14 Icosahedron

▷ `Icosahedron()` (operation)

Returns: IsPolytope

Returns the icosahedron, `{3, 5}`.

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

5.1.15 HemiIcosahedron

▷ `HemiIcosahedron()` (operation)

Returns: IsPolytope

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

Example

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

5.1.16 SmallStellatedDodecahedron

▷ `SmallStellatedDodecahedron()` (operation)

Returns: IsPolytope

Constructs the small stellated dodecahedron combinatorially. This is the same combinatorial object as the great dodecahedron. You may also use the command `GreatDodecahedron();`.

Example

```
gap> SmallStellatedDodecahedron()=GreatDodecahedron();
true
gap> Size(GreatDodecahedron());
120
```

5.1.17 24Cell

▷ `24Cell()` (operation)

Returns: IsPolytope

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

Example

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

5.1.18 Hemi24Cell

▷ `Hemi24Cell()` (operation)

Returns: IsPolytope

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

Example

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

5.1.19 120Cell

▷ `120Cell()` (operation)

Returns: IsPolytope

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

Example

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

5.1.20 Hemi120Cell

▷ `Hemi120Cell()` (operation)

Returns: IsPolytope

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

Example

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

5.1.21 600Cell

▷ `600Cell()` (operation)

Returns: IsPolytope

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

Example

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

5.1.22 Hemi600Cell

▷ `Hemi600Cell()` (operation)

Returns: IsPolytope

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

Example

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

5.1.23 BrucknerSphere

▷ `BrucknerSphere()` (operation)

Returns: IsPoset

Returns Bruckner's sphere.

Example

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

5.1.24 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 ]
```

5.1.25 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 ]
```

5.1.26 GrandAntiprism

▷ GrandAntiprism() (operation)

Returns: IsPolytope

Returns the Grand Antiprism.

5.1.27 STG1 (for IsInt)

▷ STG1(*n*) (operation)

Returns: premaniplex

Builds the 1 flag premaniplex of rank *n*. See VOLTAGE OPERATIONS ON MANIPLEXES (citation coming soon).

Example

```
gap> STG1(5);
Premaniplex of rank 5 with 1 flag
```

5.1.28 STG2 (for IsInt,IsList)

▷ STG2(*n*, *I*) (operation)

Returns: premaniplex

Builds the 2 flag premaniplex of rank *n* with semi-edges in *I*. See VOLTAGE OPERATIONS ON MANIPLEXES (citation coming soon).

Example

```
gap> STG2(5,[2,4]);
Premaniplex of rank 5 with 2 flags
```

5.1.29 STG3 (for IsInt,IsInt)

▷ STG3(*n*, *i*) (operation)

Returns: premaniplex

Builds the 3 flag premaniplex of rank *n* described on Page 11 of Symmetry Type Graphs of Polytopes and Maniplexes [CDRFHT15] (https://doi.org/10.1007/s00026-015-0263-z). The edges of label i-1 and i+1 are parallel.

Example

```
gap> STG3(5,2);
Premaniplex of rank 5 with 3 flags
```

5.1.30 STG3 (for IsInt,IsInt,IsInt)

▷ `STG3(n, i, j)` (operation)

Returns: premaniplex

Assumes `j=i+1` and builds the 3 flag premaniplex of rank *n* described on Page 11 of Symmetry Type Graphs of Polytopes and Maniplexes [CDRFHT15] (`https://doi.org/10.1007/s00026-015-0263-z`). There are edges of label *i* and *j*.

Example
```
gap> STG3(6,2,3);
Premaniplex of rank 6 with 3 flags
```

5.2 Flat and tight polytopes

5.2.1 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.

Example
```
gap> FlatOrientablyRegularPolyhedron(4,2,3,3);
FlatOrientablyRegularPolyhedron(4,2,-1,1)
```

5.2.2 FlatOrientablyRegularPolyhedraOfType (for IsList)

▷ `FlatOrientablyRegularPolyhedraOfType(sym)` (operation)

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

Example
```
ap> FlatOrientablyRegularPolyhedraOfType([6,6]);
[ FlatOrientablyRegularPolyhedron(6,6,3,1), FlatOrientablyRegularPolyhedron(6,6,-1,1),
  FlatOrientablyRegularPolyhedron(6,6,-1,3) ]
```

5.2.3 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*).

Example
```
gap> TightOrientablyRegularPolytopesOfType([6,6]);
[ FlatOrientablyRegularPolyhedron(6,6,3,1), FlatOrientablyRegularPolyhedron(6,6,-1,1),
  FlatOrientablyRegularPolyhedron(6,6,-1,3) ]
```

5.3 The Tomotope

5.3.1 Tomotope

▷ `Tomotope()` (operation)

Returns: maniplex

Constructs the *Tomotope* from [MPW12]

Example

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

5.4 Toroids

5.4.1 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
```

5.4.2 ToroidalMap36

▷ `ToroidalMap36(u[, v])` (function)

Returns: IsManiplex

Returns the toroidal map $\{3,6\}_{\vec{u},\vec{v}}$. If only `u` is given, then we return the corresponding reflexible map (if `u` is [a, 0] or [a, a]) or chiral map.

Example

```
gap> Size(ToroidalMap36([3,0])) = 108;
true
gap> SmallestReflexibleCover(ToroidalMap36([2,3])) = ToroidalMap36([19,0]);
true
gap> ToroidalMap36([3,0]) = ToroidalMap36([3,0],[0,3]);
true
gap> ToroidalMap36([2,3]) = ToroidalMap36([2,3],[-3,5]);
```

```
true
gap> NumberOfFlagOrbits(ToroidalMap36([3,0],[-2,4]));
3
gap> NumberOfFlagOrbits(ToroidalMap36([4,3],[5,0]));
6
```

5.4.3 ToroidalMap63

▷ ToroidalMap63(*u*[, *v*]) (function)

Returns: IsManiplex

Returns the toroidal map $\{6,3\}_{\vec{u},\vec{v}}$. If only `u` is given, then we return the corresponding reflexible map (if `u` is [a, 0] or [a, a]) or chiral map.

Example
```
gap> Size(ToroidalMap63([3,0])) = 108;
true
gap> SmallestReflexibleCover(ToroidalMap63([2,3])) = ToroidalMap63([19,0]);
true
gap> ToroidalMap63([3,0]) = ToroidalMap63([3,0],[0,3]);
true
gap> ToroidalMap63([2,3]) = ToroidalMap63([2,3],[-3,5]);
true
gap> NumberOfFlagOrbits(ToroidalMap63([3,0],[-2,4]));
3
gap> NumberOfFlagOrbits(ToroidalMap63([4,3],[5,0]));
6
```

5.4.4 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
```

5.4.5 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.

Example
```
gap> CubicToroid(2,[[2,0],[0,2]]);
3-maniplex
gap> last=ToroidalMap44([2,0]);
true
```

5.4.6 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 1 or 2.

Example
```
gap> M := 3343Toroid(3,1);
ReflexibleManiplex([ 3, 3, 4, 3 ], "(r0 r1 r2 r3 r2 r1 r4 r3 r2 r3 r4 r1 r2 r3 r2 r1)^3")
gap> IsPolytopal(M);
true
gap> IsPolytopal(3343Toroid(1,1));
false
```

5.4.7 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 1 or 2.

Example
```
gap> M := 24CellToroid(3,1);;
gap> Dual(M) = 3343Toroid(3,1);
true
```

5.5 Uniform and Archimedean polyhedra

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

5.5.1 Cuboctahedron

▷ `Cuboctahedron()` (operation)

Returns: maniplex

Constructs the cuboctahedron.

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

5.5.2 TruncatedTetrahedron

▷ `TruncatedTetrahedron()` (operation)

Returns: maniplex

Constructs the truncated tetrahedron.

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

5.5.3 TruncatedOctahedron

▷ `TruncatedOctahedron()` (operation)

Returns: maniplex

Constructs the truncated octahedron.

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

5.5.4 TruncatedCube

▷ `TruncatedCube()` (operation)

Returns: maniplex

Constructs the truncated octahedron.

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

5.5.5 Icosadodecahedron

▷ `Icosadodecahedron()` (operation)

Returns: maniplex

Constructs the icosadodecahedron.

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

5.5.6 TruncatedIcosahedron

▷ `TruncatedIcosahedron()` (operation)

Returns: maniplex

Constructs the truncated icosahedron.

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

5.5.7 SmallRhombicuboctahedron

▷ `SmallRhombicuboctahedron()` (operation)

Returns: maniplex

Constructs the small rhombicuboctahedron.

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

5.5.8 Pseudorhombicuboctahedron

▷ `Pseudorhombicuboctahedron()` (operation)

Returns: maniplex

Constructs the pseudorhombicuboctahedron.

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

5.5.9 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
```

5.5.10 SmallRhombicosidodecahedron

▷ `SmallRhombicosidodecahedron()` (operation)

Returns: maniplex

Constructs the small rhombicosidodecahedron.

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

5.5.11 GreatRhombicosidodecahedron

▷ `GreatRhombicosidodecahedron()` (operation)

Returns: maniplex

Constructs the great rhombicosidodecahedron.

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

5.5.12 SnubDodecahedron

▷ `SnubDodecahedron()` (operation)

Returns: maniplex

Constructs the small snub dodecahedron.

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

5.5.13 TruncatedDodecahedron

▷ `TruncatedDodecahedron()` (operation)

Returns: maniplex

Constructs the truncated dodecahedron.

Example
```
gap> Facets(TruncatedDodecahedron());
[ Pgon(10), Pgon(3) ]
```

5.5.14 GreatRhombicuboctahedron

▷ `GreatRhombicuboctahedron()` (operation)

Returns: maniplex

Constructs the great rhombicuboctahedron.

Example
```
gap> Size(ConnectionGroup(GreatRhombicuboctahedron()));
5308416
```

Chapter 6

Combinatorial Structure of Maniplexes

6.1 Basics

6.1.1 Size (for IsPremaniplex)

▷ `Size(M)` (attribute)

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

Synonym: `NumberOfFlags`.

6.1.2 RankManiplex (for IsPremaniplex)

▷ `RankManiplex(M)` (attribute)

Returns: The rank of the premaniplex *M*.

6.2 Faces

6.2.1 NumberOfIFaces (for IsManiplex, IsInt)

▷ `NumberOfIFaces(M, i)` (operation)

Returns The number of *i*-faces of *M*.

Example

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

6.2.2 NumberOfVertices (for IsManiplex)

▷ `NumberOfVertices(M)` (attribute)

Returns the number of vertices of *M*.

Example

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

6.2.3 NumberOfEdges (for IsManiplex)

▷ `NumberOfEdges(M)` (attribute)

Returns the number of edges of *M*.

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

6.2.4 NumberOfFacets (for IsManiplex)

▷ `NumberOfFacets(M)` (attribute)

Returns the number of facets of *M*.

```
gap> NumberOfFacets(Bk21(4,6));
4
```

6.2.5 NumberOfRidges (for IsManiplex)

▷ `NumberOfRidges(M)` (attribute)

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

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

6.2.6 Fvector (for IsManiplex)

▷ `Fvector(M)` (attribute)

Returns the f-vector of *M*.

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

6.2.7 Section(s)

▷ `Section(M, j, i)` (operation)
▷ `Section(M, j, i, k)` (operation)
▷ `Sections(M, j, i)` (operation)

`Section(M,j,i)` 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. `Section(M,j,i,k)` 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. `Sections(M,j,i)` returns all sections of type `F_j / F_i`, where `F_j` is a *j*-face and `F_i` is an incident *i*-face.

Example
```
gap> Section(ToroidalMap44([2,2]),3,0);
Pgon(4)
gap> Section(Cuboctahedron(),2,-1,1);
Pgon(4)
gap> Section(Cuboctahedron(),2,-1,4);
Pgon(3)
gap> Sections(Cuboctahedron(),2,-1);
[ Pgon(4), Pgon(3) ]
```

6.2.8 Facet(s)

▷ `Facets(M)` (attribute)
▷ `Facet(M, k)` (operation)
▷ `Facet(M)` (attribute)

Returns the facet-types of `M` (i.e. the maniplexes corresponding to the facets). Returns the facet of `M` that contains the flag number `k` (that is, the maniplex corresponding to the facet). Returns the facet of `M` that contains flag number 1 (that is, the maniplex corresponding to the facet).

Example
```
gap> Facets(Cuboctahedron());
[ Pgon(4), Pgon(3) ]
gap> Facet(Cuboctahedron(),4);
Pgon(3)
gap> Facet(Cuboctahedron());
Pgon(4)
```

6.2.9 Vertex Figure(s)

▷ `VertexFigures(M)` (attribute)
▷ `VertexFigure(M, k)` (operation)
▷ `VertexFigure(M)` (attribute)

Returns the types of vertex-figures of `M` (i.e. the maniplexes corresponding to the vertex-figures). Returns the vertex-figure of `M` that contains flag number `k`. Returns the vertex-figure of `M` that contains the base flag.

Example
```
gap> p:=Dual(SmallRhombicosidodecahedron());
Dual(3-maniplex)
gap> VertexFigures(p);
[ Pgon(5), Pgon(4), Pgon(3) ]
gap> VertexFigure(p,4);
Pgon(4)
gap> VertexFigure(p);
Pgon(5)
```

6.2.10 VertDegrees (for IsManiplex)

▷ VertDegrees(*M*) (attribute)

Returns: IsList

Returns a list that describes how many vertices *M* has of each valency. This list has the form [[v1, n1], [v2, n2], ...] to indicate that there are n1 vertices of valcency v1, etc.

Example
```
gap> VertDegrees(Pyramid(5));
[ [3, 5], [5, 1] ];
gap> VertDegrees(Kis(Cube(3)));
[ [4, 6], [6, 8] ];
gap> VertDegrees(Prism(5));
[ [3, 10] ]
```

6.2.11 FaceSizes (for IsManiplex)

▷ FaceSizes(*M*) (attribute)

Returns: IsList

Returns a list that describes how many 2-faces *M* has of each size. This list has the form [[f1, n1], [f2, n2], ...] to indicate that there are n1 f1-gonal faces, etc.

Example
```
gap> FaceSizes(Cube(3));
[ [ 4, 6 ] ]
gap> FaceSizes(Cube(4));
[ [ 4, 24 ] ]
gap> FaceSizes(Prism(Dodecahedron()));
[ [ 4, 30 ], [ 5, 24 ] ]
```

6.2.12 FacetList (for IsManiplex)

▷ FacetList(*M*) (attribute)

Returns: list

Lists the facets of the maniplex *M* as lists of flags.

Example
```
gap> m:=Cuboctahedron();
3-maniplex
gap> FacetList(m);
[ [ 1, 2, 3, 5, 7, 10, 13, 18 ], [ 4, 6, 9, 12, 16, 21 ], [ 8, 14, 15, 24, 25, 34 ],
[ 11, 19, 20, 29, 30, 39 ], [ 17, 26, 27, 36, 37, 46, 47, 57 ], [ 22, 31, 32, 41, 42, 52, 53, 62
[ 23, 28, 33, 38, 43, 49 ], [ 35, 44, 45, 55, 56, 65, 66, 75 ],
[ 40, 50, 51, 60, 61, 70, 71, 80 ], [ 48, 54, 59, 64, 69, 74 ], [ 58, 67, 68, 77, 78, 86 ],
[ 63, 72, 73, 82, 83, 89 ], [ 76, 81, 85, 88, 91, 93 ], [ 79, 84, 87, 90, 92, 94, 95, 96 ] ]
```

6.2.13 VertexList (for IsManiplex)

▷ VertexList(*M*) (attribute)

Returns: list

Lists the vertices of the maniplex *M* as lists of flags.

Example
```
gap> m:=Cuboctahedron();
3-maniplex
gap> VertexList(m);
[ [ 1, 3, 4, 8, 9, 15, 17, 26 ], [ 2, 5, 6, 11, 12, 20, 22, 31 ], [ 7, 13, 14, 23, 24, 33, 35, 44
  [ 10, 18, 19, 28, 29, 38, 40, 50 ], [ 16, 21, 27, 32, 37, 42, 48, 54 ],
  [ 25, 34, 36, 45, 46, 56, 58, 67 ], [ 30, 39, 41, 51, 52, 61, 63, 72 ],
  [ 43, 49, 55, 60, 65, 70, 76, 81 ], [ 47, 57, 59, 68, 69, 78, 79, 87 ],
  [ 53, 62, 64, 73, 74, 83, 84, 90 ], [ 66, 75, 77, 85, 86, 91, 92, 95 ],
  [ 71, 80, 82, 88, 89, 93, 94, 96 ] ]
```

6.2.14 NFacesList (for IsManiplex,IsInt)

▷ `NFacesList(M, n)` (operation)

Returns: list

Lists the `n`-faces of the maniplex `M` as lists of flags.

Example
```
gap> m:=Cuboctahedron();
3-maniplex
gap> NFacesList(m);
gap> m:=Cuboctahedron();
3-maniplex
gap> NFacesList(m,2)=FacetList(m);
true
gap> NFacesList(m,1);
[ [ 1, 2, 4, 6 ], [ 3, 7, 8, 14 ], [ 5, 10, 11, 19 ], [ 9, 16, 17, 27 ], [ 12, 21, 22, 32 ],
  [ 13, 18, 23, 28 ], [ 15, 25, 26, 36 ], [ 20, 30, 31, 41 ], [ 24, 34, 35, 45 ],
  [ 29, 39, 40, 51 ], [ 33, 43, 44, 55 ], [ 37, 47, 48, 59 ], [ 38, 49, 50, 60 ],
  [ 42, 53, 54, 64 ], [ 46, 57, 58, 68 ], [ 52, 62, 63, 73 ], [ 56, 66, 67, 77 ],
  [ 61, 71, 72, 82 ], [ 65, 75, 76, 85 ], [ 69, 74, 79, 84 ], [ 70, 80, 81, 88 ],
  [ 78, 86, 87, 92 ], [ 83, 89, 90, 94 ], [ 91, 93, 95, 96 ] ]
```

6.3 Flatness

6.3.1 Flatness

▷ `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.

Example
```
gap> IsFlat(HemiCube(3));
true
gap> IsFlat(Simplex(3),0,2);
false
```

6.4 Schlafli symbol

6.4.1 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.

Example
```
gap> SchlafliSymbol(SmallRhombicosidodecahedron());
[ [ 3, 4, 5 ], 4 ]
```

6.4.2 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.

Example
```
gap> M := ReflexibleManiplex([4,4], "(r0 r1)^2");;
gap> PseudoSchlafliSymbol(M);
[4, 4]
gap> SchlafliSymbol(M);
[2, 4]
gap> PseudoSchlafliSymbol(M);
[2, 4]
```

6.4.3 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).

Example
```
gap> IsEquivelar(Bk2l(6,18));
true
```

6.4.4 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.

Example
```
gap> F := FreeGroup("s0","s1","s2","s3");;
gap> s0 := F.1;;  s1 := F.2;;  s2 := F.3;;  s3 := F.4;;
```

```
gap> rels := [ s0*s0, s1*s1, s2*s2, s3*s3, s0*s2*s0*s2,
> s1*s2*s1*s2, s0*s3*s0*s3, s1*s3*s1*s3,
> s2*s3*s2*s3*s2*s3*s2*s3, s0*s1*s0*s1*s0*s1*s0*s1*s0*s1 ];;
gap> poly := F / rels;;
gap> IsDegenerate(ARP(poly));
true
```

6.4.5 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.

Example

```
gap> IsTight(ToroidalMap44([2,0]));
true
```

6.4.6 EulerCharacteristic (for IsManiplex)

▷ `EulerCharacteristic(M)` (attribute)

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

Example

```
gap> EulerCharacteristic(Bk2lStar(3,5));
-10
```

6.4.7 Genus (for IsManiplex)

▷ `Genus(M)` (attribute)

Returns: The genus of the given 3-maniplex.

Example

```
gap> Genus(Bk2lStar(3,5));
6
```

6.4.8 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
```

6.4.9 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
```

6.4.10 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
```

6.4.11 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
```

6.5 Orientability

6.5.1 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
```

6.5.2 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
```

6.5.3 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
```

6.5.4 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
```

6.5.5 OrientableCover (for IsManiplex)

▷ OrientableCover(*M*) (attribute)

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

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

6.5.6 IOrientableCover (for IsManiplex, IsList)

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

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

Example
```
gap> SchlafliSymbol(IOrientableCover(Cube(3), [2]));
[ 4, 6 ]
```

6.6 Zigzags and holes

6.6.1 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.

Example
```
gap> ZigzagLength(Cube(3),1);
6
gap> ZigzagLength(Cube(3),2);
6
gap> ZigzagLength(Cube(3),3);
2
```

6.6.2 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.

Example
```
gap> ZigzagVector(Pseudorhombicuboctahedron());
[ [ 40, 56 ], [ 8, 32 ] ]
```

6.6.3 PetrieLength (for IsManiplex)

▷ `PetrieLength(M)` (attribute)

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

Example
```
gap> PetrieLength(Cube(3));
6
```

6.6.4 PetrieRelation (for IsInt, IsInt)

▷ `PetrieRelation(i, j)` (operation)

Returns: relation

Returns the Petrie relation for a rank *i* maniplex of length *j*.

Example
```
gap> p:=PetrieRelation(3,3);
"(r0r1r2)^3"
gap> q:=Cube(3)/p;
3-maniplex
gap> Size(q);
24
```

6.6.5 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.

Example
```
gap> HoleLength(ToroidalMap44([3,0]),2);
3
```

6.6.6 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.

Example
```
gap> HoleVector(ToroidalMap44([3,0],[0,5]));
[ [ 3, 5 ] ]
```

Chapter 7

Algebraic Structure of Maniplexes

7.1 Groups of Maps, Polytopes, and Maniplexes

7.1.1 Automorphism Groups

▷ `AutomorphismGroup(M)` (attribute)
▷ `AutomorphismGroupFpGroup(M)` (attribute)
▷ `AutomorphismGroupPermGroup(M)` (attribute)
▷ `AutomorphismGroupOnFlags(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, or AutomorphismGroupFpGroup. Returns the automorphism group of `M` as a finitely presented group. If `M` is reflexible, then this is guaranteed to be the standard presentation. Returns the automorphism group of `M` as a permutation group. This group is not guaranteed to be in any particular form. For particular permutation representations you should consider the various AutomorphismGroupOn... functions. Returns the automorphism group of `M` as a permutation group action on the flags of `M`.

Example
```
gap> s0 := (3,7)(4,8)(5,6);;
gap> s1 := (2,3)(4,6)(5,7);;
gap> s2 := (1,2)(3,6)(4,8)(5,7);;
gap> poly := Group([s0,s1,s2]);;
gap> p:=ARP(poly);
regular 3-polytope
gap> AutomorphismGroup(p);
Group([ (3,7)(4,8)(5,6), (2,3)(4,6)(5,7), (1,2)(3,6)(4,8)(5,7) ])
gap> AutomorphismGroupFpGroup(p);
<fp group on the generators [ r0, r1, r2 ]>
gap> AutomorphismGroupPermGroup(Cube(3));
Group([ (3,4), (2,3)(4,5), (1,2)(5,6) ])
gap> AutomorphismGroupOnFlags(Cube(3));
<permutation group with 3 generators>
gap> GeneratorsOfGroup(last);
[ (1,20)(2,13)(3,10)(4,34)(5,35)(6,7)(8,27)(9,28)(11,23)(12,24)(14,44)(15,45)(16,18)(17,19)(21,40
  (1,11)(2,18)(3,4)(5,26)(6,41)(7,8)(9,33)(10,45)(12,15)(13,31)(14,25)(16,28)(17,27)(19,22)(20,38
```

```
  (1,3)(2,7)(4,25)(5,28)(6,13)(8,32)(9,35)(10,20)(11,14)(12,17)(15,47)(16,40)(18,21)(19,24)(22,48
```

7.1.2 ConnectionGroup (for IsPremaniplex)

▷ `ConnectionGroup(M)` (attribute)

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

Example

```
gap> ConnectionGroup(HemiCube(3));
Group([ (1,8)(2,7)(3,14)(4,13)(5,20)(6,19)(9,16)(10,15)(11,22)(12,21)(17,24)(18,23), (1,3)(2,5)
  (4,6)(7,9)(8,11)(10,12)(13,15)(14,17)(16,18)(19,21)(20,23)(22,24), (1,2)(3,4)(5,6)(7,8)(9,10)
  (11,12)(13,14)(15,16)(17,18)(19,20)(21,22)(23,24) ])
```

7.1.3 EvenConnectionGroup (for IsManiplex)

▷ `EvenConnectionGroup(M)` (attribute)

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

Example

```
gap> EvenConnectionGroup(HemiCube(3));
Group([ (1,11,24,14)(2,9,18,20)(3,17,22,8)(4,15,12,19)(5,23,16,7)(6,21,10,13), (1,4,5)(2,6,3)
  (7,10,11)(8,12,9)(13,16,17)(14,18,15)(19,22,23)(20,24,21) ])
```

7.1.4 RotationGroup (for IsManiplex)

▷ `RotationGroup(M)` (attribute)

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

Example

```
gap> RotationGroup(HemiCube(3));
Group([ r0*r1, r1*r2 ])
```

7.1.5 RotationGroupFpGroup (for IsManiplex)

▷ `RotationGroupFpGroup(M)` (attribute)

Returns the rotation group of `M`, as a finitely presented group on the standard generators.

Example

```
gap> RotationGroupFpGroup(ToroidalMap44([1,2]));
<fp group on the generators [ s1, s2 ]>
gap> RelatorsOfFpGroup(last);
[ (s1*s2)^2, s1^4, s2^4, s2^-1*s1*(s2*s1^-1)^2 ]
```

7.1.6 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.

Example
```
gap> M := ToroidalMap44([1,2]);
ToroidalMap44([ 1, 2 ])
gap> G := ChiralityGroup(M);
Group([ s2^-1*s1^-1*s2*s1^3*s2*s1 ])
gap> Size(G);
5
```

7.1.7 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.

Example
```
gap> ExtraRelators(HemiCube(3));
[ (r0*r1*r2)^3 ]
```

7.1.8 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.

Example
```
gap> ExtraRotRelators(HemiCube(3));
[ (F2^-1*F1^-1)^2, (F2*F1^2*F2^-1*F1^-1)^2 ]
```

7.1.9 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.

7.2 Automorphism group acting on faces and chains

7.2.1 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. If the automorphism group has a standard set of generators in a standard order, then the output is generated by the action of those generators.

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) ])
```

7.2.2 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. If the automorphism group has a standard set of generators in a standard order, then the output is generated by the action of those generators.

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

7.2.3 AutomorphismGroupOnVertices (for IsManiplex)

▷ `AutomorphismGroupOnVertices(M)` (attribute)

Returns: IsPermGroup

Returns a permutation group, representing the action of AutomorphismGroup(M) on the vertices of M. If the automorphism group has a standard set of generators in a standard order, then the output is generated by the action of those generators.

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) ])
```

7.2.4 AutomorphismGroupOnEdges (for IsManiplex)

▷ `AutomorphismGroupOnEdges(M)` (attribute)

Returns: IsPermGroup

Returns a permutation group, representing the action of AutomorphismGroup(M) on the edges of M. If the automorphism group has a standard set of generators in a standard order, then the output is generated by the action of those generators.

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) ])
```

7.2.5 AutomorphismGroupOnFacets (for IsManiplex)

▷ `AutomorphismGroupOnFacets(M)` (attribute)

Returns: IsPermGroup

Returns a permutation group, representing the action of AutomorphismGroup(M) on the facets of M. If the automorphism group has a standard set of generators in a standard order, then the output is generated by the action of those generators.

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

7.3 Number of orbits and transitivity

7.3.1 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
```

7.3.2 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
```

7.3.3 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
```

7.3.4 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
```

7.3.5 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
```

7.3.6 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
```

7.3.7 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
```

7.3.8 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
```

7.3.9 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
```

7.3.10 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
```

7.3.11 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
```

7.4 Faithfulness

7.4.1 IsVertexFaithful (for IsManiplex)

▷ 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
```

7.4.2 IsFacetFaithful (for IsManiplex)

▷ 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
```

7.4.3 MaxVertexFaithfulQuotient (for IsManiplex)

▷ `MaxVertexFaithfulQuotient(M)` (operation)

Returns: Q

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

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

7.5 Flag orbits

7.5.1 Flags (for IsPremaniplex)

▷ `Flags(M)` (attribute)

Returns: IsList

The list of flags of the premaniplex *M*.

Example
```
gap> Flags(Pgon(5));
[ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 ]
gap> M := Maniplex(Group((3,4)(5,6)(7,8)(9,10), (3,6)(4,5)(7,10)(8,9), (3,7)(4,8)(5,9)(6,10)));;
gap> Flags(M);
[ 3, 4, 5, 6, 7, 8, 9, 10 ]
```

7.5.2 BaseFlag (for IsPremaniplex)

▷ `BaseFlag(M)` (attribute)

Returns: IsObject

The base flag of the premaniplex *M*. By default, when the set of flags is a set of positive integers, the base flag is the minimum of the set of flags.

Example
```
gap> BaseFlag(Cube(3));
1
gap> M := Maniplex(Group((3,4)(5,6)(7,8)(9,10), (3,6)(4,5)(7,10)(8,9), (3,7)(4,8)(5,9)(6,10)));;
gap> BaseFlag(M);
3
```

7.5.3 SymmetryTypeGraph (for IsPremaniplex)

▷ `SymmetryTypeGraph(M[, A])` (attribute)

Returns: IsPremaniplex

Returns the Symmetry Type Graph of the premaniplex *M* with respect to the subgroup *A* of the automorphism group; that is, the quotient of the flag graph of *M* by *A*. If *A* is not included, then returns the Symmetry Type Graph relative to the whole automorphism group of *M*.

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

```
gap> M:=Cube(3);;
gap> A:=AutomorphismGroupOnFlags(M);;
gap> B:=Group(A.1,A.2*A.3);;
gap> SymmetryTypeGraph(M,B);
Edge labeled graph with 2 vertices, and edge labels [ 0, 1, 2 ]
```

7.5.4 NumberOfFlagOrbits (for IsPremaniplex)

▷ `NumberOfFlagOrbits(M)` (attribute)

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

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

7.5.5 FlagOrbitRepresentatives (for IsPremaniplex)

▷ `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 ]
```

7.5.6 FlagOrbitsStabilizer (for IsPremaniplex)

▷ `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
```

7.5.7 IsReflexible (for IsPremaniplex)

▷ `IsReflexible(M)` (property)

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

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

7.5.8 IsChiral (for IsPremaniplex)

▷ IsChiral(*M*) (property)

Returns: Whether the premaniplex *M* is chiral.

Example

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

7.5.9 IsRotary (for IsPremaniplex)

▷ 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
```

7.5.10 FlagOrbits (for IsPremaniplex)

▷ 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,
```

Chapter 8

Comparing maniplexes

8.1 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 ParseGgiRels that underlies many quotient operations.

8.1.1 IsQuotient

▷ `IsQuotient(M1, M2)` (operation)

▷ `IsQuotient(g, h)` (operation)

Returns: `IsBool`

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

Example
```
gap> IsQuotient(Cube(3),HemiCube(3));
true
gap> IsQuotient(UniversalSggi([4,3]),AutomorphismGroup(HemiCube(3)));
true
```

8.1.2 IsRootedQuotient (for IsManiplex, IsManiplex, IsInt, IsInt)

▷ `IsRootedQuotient(M1, M2, i, j)` (operation)

Returns: `IsBool`

Returns whether there is a maniplex homomorphism from `M1` to `M2` that sends flag `i` of `M1` to flag `j` of `M2`.

Example
```
gap> IsRootedQuotient(Pyramid(8), Pyramid(4), 1, 1);
true
```

```
gap> IsRootedQuotient(Pyramid(8), Pyramid(4), 1, 2);
false
```

8.1.3 IsRootedQuotient (for IsManiplex, IsManiplex)

▷ `IsRootedQuotient(M1, M2)` (operation)

Returns: `IsBool`

Returns whether there is a maniplex homomorphism from *M1* to *M2* that sends `BaseFlag(M1)` to `BaseFlag(M2)`.

Example
```
gap> IsRootedQuotient(ToroidalMap44([4,4]), ToroidalMap44([4,0]));
true
gap> IsRootedQuotient(ToroidalMap44([1,2]), ToroidalMap44([2,1]));
false
```

8.1.4 IsCover (for IsPremaniplex, IsPremaniplex)

▷ `IsCover(M1, M2)` (operation)

Returns: `IsBool`

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

Example
```
gap> IsCover(HemiDodecahedron(),Dodecahedron());
true
```

8.1.5 IsCover (for IsSggi, IsSggi)

▷ `IsCover(g, h)` (operation)

Returns: `IsBool`

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

Example
```
gap> IsCover(HemiCube(3), Cube(3));
true
gap> IsCover(AutomorphismGroup(HemiCube(3)), UniversalSggi([4,3]));
true
```

8.1.6 IsRootedCover (for IsManiplex, IsManiplex, IsInt, IsInt)

▷ `IsRootedCover(M1, M2, i, j)` (operation)

Returns: `IsBool`

Returns whether there is a maniplex homomorphism from *M2* to *M1* that sends flag *j* of *M2* to flag *i* of *M1*.

Example
```
gap> IsRootedCover(Pyramid(4), Pyramid(8), 1, 1);
true
gap> IsRootedCover(Pyramid(4), Pyramid(8), 1, 2);
false
```

8.1.7 IsRootedCover (for IsManiplex, IsManiplex)

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

Returns: IsBool

Returns whether there is a maniplex homomorphism from *M2* to *M1* that sends BaseFlag(M2) to BaseFlag(M1).

```
gap> IsRootedCover(ToroidalMap44([4,0]), ToroidalMap44([4,4]));
true
gap> IsRootedCover(ToroidalMap44([1,2]), ToroidalMap44([2,1]));
false
```

8.1.8 IsIsomorphicManiplex (for IsManiplex, IsManiplex)

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

Returns: IsBool

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

```
gap> IsIsomorphicManiplex(HemiCube(3),Petrial(Simplex(3)));
true
```

8.1.9 IsIsomorphicRootedManiplex (for IsManiplex, IsManiplex, IsInt, IsInt)

▷ IsIsomorphicRootedManiplex(*M1*, *M2*, *i*, *j*) (operation)

Returns: IsBool

Returns whether there is an isomorphism from *M1* to *M2* that sends flag *j* of *M2* to flag *i* of *M1*.

```
gap> IsIsomorphicManiplex(ToroidalMap44([1,2]), ToroidalMap44([2,1]));
true
gap> FlagOrbitRepresentatives(ToroidalMap44([1,2]));
[1, 21]
gap> IsIsomorphicRootedManiplex(ToroidalMap44([1,2]), ToroidalMap44([1,2]), 1, 1);
true
gap> IsIsomorphicRootedManiplex(ToroidalMap44([1,2]), ToroidalMap44([1,2]), 1, 21);
false
gap> IsIsomorphicRootedManiplex(ToroidalMap44([1,2]), ToroidalMap44([2,1]), 1, 1);
false
```

8.1.10 IsIsomorphicRootedManiplex (for IsManiplex, IsManiplex)

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

Returns: IsBool

Returns whether there is an isomorphism from *M1* to *M2* that sends BaseFlag(M2) to BaseFlag(M1).

```
gap> IsIsomorphicManiplex(ToroidalMap44([1,2]), ToroidalMap44([2,1]));
true
gap> IsIsomorphicRootedManiplex(ToroidalMap44([1,2]), ToroidalMap44([2,1]));
false
```

```
gap> IsIsomorphicRootedManiplex(ToroidalMap44([1,2]), EnantiomorphicForm(ToroidalMap44([2,1])));
true
```

8.1.11 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*.

Example
```
gap> SmallestReflexibleCover(ToroidalMap44([2,3],[3,2]));
reflexible 3-maniplex
gap> last=ToroidalMap44([5,0]);
true
```

8.1.12 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".

Example
```
gap> q:=QuotientManiplex(CubicTiling(2),"(r0 r1 r2 r1)^5, (r1 r0 r1 r2)^2");
3-maniplex
gap> SchlafliSymbol(q);
[ 4, 4 ]
```

8.1.13 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.

Example
```
gap> q:=ReflexibleQuotientManiplex(CubicTiling(2),"(r0 r1 r2 r1)^5, (r1 r0 r1 r2)^2");
reflexible 3-maniplex with 8 flags
gap> last=ToroidalMap44([1,0]);
true
```

8.1.14 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 ParseGgiRels.

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
```

8.1.15 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
```

8.1.16 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 9

Polytope Constructions and Operations

9.1 Extensions, amalgamations, and quotients

9.1.1 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
```

9.1.2 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
```

9.1.3 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
```

9.1.4 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
```

9.1.5 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
```

9.1.6 Amalgamate (for IsManiplex, IsManiplex)

▷ `Amalgamate(M1, M2)` (operation)

Returns: IsManiplex

Constructs the amalgamation of the n-maniplexes *M1* and *M2*. This does not check whether *M1* and *M2* are compatible, so the output may not have facets isomorphic to *M1* or vertex-figures isomorphic to *M2*. Currently only defined for rotary maniplexes. Note that if M1 and M2 are chiral, then the amalgamation depends on the choices of base flag for each.

Example
```
gap> Amalgamate(Cube(3), Simplex(3)) = Cube(4);
true
gap> Size(Amalgamate(ToroidalMap44([1,2]), Cube(3)));
240
gap> Size(Amalgamate(ToroidalMap44([1,2]), ToroidalMap44([2,1])));
240
gap> Size(Amalgamate(ToroidalMap44([1,2]), ToroidalMap44([1,2])));
4
```

9.1.7 Medial (for IsManiplex)

▷ `Medial(M)` (operation)

Returns: IsManiplex

Given a 3-maniplex *M*, returns its medial.

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

9.2 Duality

9.2.1 Dual (for IsManiplex)

▷ Dual(*M*) (operation)

Returns: The maniplex that is dual to *M*.

Example

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

9.2.2 IsSelfDual (for IsManiplex)

▷ IsSelfDual(*M*) (property)

Returns: Whether this maniplex is isomorphic to its dual.

Also works for IsPoset objects.

Example

```
gap> IsSelfDual(Cube(3));
false
gap> IsSelfDual(Simplex(5));
true
```

9.2.3 IsInternallySelfDual (for IsManiplex)

▷ 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*. You can also input a string for *x*, in which case it is converted to `SggiElement(M, x)`. This property is only defined for rotary (chiral or reflexible) maniplexes.

Example

```
gap> IsInternallySelfDual(Simplex(4));
true
gap> IsInternallySelfDual(Simplex(3), "r0")
#I  The given automorphism is not dualizing; searching for another...
true
gap> IsInternallySelfDual(Simplex(3), "r0 r1 r2 r0 r1 r0");
true
gap> IsInternallySelfDual(ToroidalMap44([6,0]));
false
gap> IsInternallySelfDual(ToroidalMap44([5,0]));
true
gap> IsInternallySelfDual(ToroidalMap44([1,2]));
false
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
```

9.2.4 IsExternallySelfDual (for IsManiplex)

▷ `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(ToroidalMap44([6,0]));
true
gap> IsExternallySelfDual(ToroidalMap44([5,0]));
false
gap> IsExternallySelfDual(Cube(3));
false
```

9.2.5 IsProperlySelfDual (for IsManiplex)

▷ `IsProperlySelfDual(M)` (property)

Returns: `true` or `false`

Returns whether this rooted maniplex is "properly self-dual", meaning that there is an isomorphism of *M* to its dual that preserves the flag-orbits. Note that all reflexible self-dual maniplexes are properly self-dual.

Example

```
gap> IsProperlySelfDual(Cube(4));
false
gap> IsProperlySelfDual(Simplex(4));
true
gap> IsProperlySelfDual(ARP([4,5,4]));
true
gap> IsProperlySelfDual(ToroidalMap44([1,2]));
false
gap> IsProperlySelfDual(RotaryManiplex([4,4,4],"(s2^-1 s1) (s2 s1^-1)^3, (s2 s3^-1) (s2^-1 s3)^3"
true
gap> IsProperlySelfDual(RotaryManiplex([4,4,4],"(s2^-1 s1)^3 (s2 s1^-1), (s2 s3^-1) (s2^-1 s3)^3"
false
```

9.2.6 IsImproperlySelfDual (for IsManiplex)

▷ `IsImproperlySelfDual(M)` (property)

Returns: `true` or `false`

Returns whether this rooted maniplex is improperly self-dual, meaning that it is self-dual, but there is no isomorphism of `M` to its dual that preserves the flag-orbits. Equivalent to IsSelfDual(M) and not(IsProperlySelfDual(M)).

Example

```
gap> IsImproperlySelfDual(Cube(4));
false
gap> IsImproperlySelfDual(Simplex(4));
false
gap> IsImproperlySelfDual(ToroidalMap44([1,2]));
true
gap> IsImproperlySelfDual(RotaryManiplex([4,4,4],"(s2^-1 s1) (s2 s1^-1)^3, (s2 s3^-1) (s2^-1 s3)
false
gap> IsImproperlySelfDual(RotaryManiplex([4,4,4],"(s2^-1 s1)^3 (s2 s1^-1), (s2 s3^-1) (s2^-1 s3)
true
```

9.2.7 Petrie Dual

▷ `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.

Synonym for the command is `PetrieDual`.

Example

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

9.2.8 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
```

9.2.9 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") ]
```

9.2.10 IsKaleidoscopic (for IsMapOnSurface)

▷ `IsKaleidoscopic(M)` (property)

Returns: `true` or `false`

Returns whether the map M with q-valent vertices is *kaleidoscopic*, meaning that for each integer e in [2..q-1] that is coprime to q, the map Hole(M, e) is isomorphic to M as a rooted map.

Example
```
gap> M := AbstractRegularPolytope([4,5], "(r0 r1 r2)^5");;
gap> ForAll([2,3,4], e -> IsIsomorphicRootedManiplex(Hole(M,e), M));
true
gap> IsKaleidoscopic(M);
true
gap> Filtered(SmallChiralPolyhedra(200), IsKaleidoscopic);
[ ]
```

9.2.11 ExponentGroup (for IsMapOnSurface)

▷ `ExponentGroup(M)` (attribute)

Given a map M with constant valency q, returns a list of those integers e in $\{1,\ldots,q-1\}$ such that M^e is isomorphic to M. That is, such that Hole(M,e) is isomorphic to M as a rooted map. Note that despite the name, this is simply a list and not a group.

Example
```
gap> ExponentGroup(ToroidalMap44([3,0]));
[ 1, 3 ]
gap> ExponentGroup(ToroidalMap44([1,2]));
[ 1 ]
gap> ExponentGroup(ReflexibleManiplex([10,10], "(r0 r1 r2)^2"));
[ 1, 3, 7, 9 ]
```

9.2.12 UpToDuality (for IsList)

▷ `UpToDuality(L)` (operation)

Given a list of maniplexes L, returns a sublist such that every maniplex in the list is unique and if a non-self-dual maniplex is in the list, then its dual is not in the list. In the case where two or more elements of L are duals of each other, we keep the earliest one.

Example
```
gap> UpToDuality([Cube(3), Simplex(3), CrossPolytope(3)]);
[ Cube(3), Simplex(3) ]
gap> UpToDuality([CrossPolytope(3), Simplex(3), Cube(3)]);
[ CrossPolytope(3), Simplex(3) ]
```

```
gap> L := SmallReflexibleManiplexes(3, [1..100]);;
gap> Size(L);
250
gap> L2 := UpToDuality(L);;
gap> Size(L2);
147
gap> Number(L, IsSelfDual);
44
```

9.3 Mixing of Maniplexes functions

9.3.1 Mix of groups

▷ `Mix(g, h)` (operation)

Returns: `IsGroup` .

Given two groups (either both permutation groups or both FpGroups), returns the mix of those groups. If g and h are permutation groups of degree m and n, respectively, then the mix is a permutation group of degree m+n.

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>
```

9.3.2 Mix (for IsPremaniplex, IsPremaniplex)

▷ `Mix(maniplex, maniplex)` (operation)

Returns: `IsReflexibleManiplex` .

Given two (pre-)maniplexes, returns their mix. For two reflexible maniplexes returns the IsReflexibleManiplex from the mix of their connection groups. In general, it returns the "flag mix" of the two maniplexes (see `FlagMix`).

9.3.3 FlagMix (for IsPremaniplex, IsPremaniplex)

▷ `FlagMix(maniplex, maniplex)` (operation)

Returns: `IsManiplex` .

Given two (pre-)maniplexes p, q this returns the (pre-)maniplex of their "flag" mix. That is, it constructs the mix of their connection groups, keeps the connected component with the base flags of p and q, and then builds a maniplex from this.

Example
```
gap> M := ToroidalMap44([1,2]);;
gap> FlagMix(M,M) = M;
true
gap> R := FlagMix(M, EnantiomorphicForm(M));
3-maniplex with 200 flags
```

```
gap> IsReflexible(R);
true
gap> R = ToroidalMap44([5,0]);
true
```

9.3.4 Comix

▷ `Comix(fpgroup, fpgroup)` (operation)
▷ `Comix(maniplex, maniplex)` (operation)

Returns: `IsReflexibleManiplex` .

Returns the comix of two Finitely Presented groups gp and gq. Given maniplexes returns the IsReflexibleManiplex from the comix of their connection groups

9.3.5 Indexed array tools

▷ `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.

Example
```
gap> LtoC(5,4,14);
[ 1, 2 ]
gap> CtoL(3,2,5,4);
8
gap> LtoC(8,5,4);
[ 3, 2 ]
```

9.4 Products

9.4.1 Pyramids

▷ `Pyramid(M)` (operation)
▷ `Pyramid(k)` (operation)

In the first form, returns the pyramid over *M*. In the second form, 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 ] ]
```

9.4.2 Prisms

▷ `Prism(M)` (operation)
▷ `Prism(k)` (operation)

In the first form, returns the prism over *M*. In the second form, returns the prism over a *k*-gon.

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

9.4.3 Antiprisms

▷ `Antiprism(M)` (operation)
▷ `Antiprism(k)` (operation)

In the first form, returns the antiprism over *M*. In the second form, returns the antiprism over a *k*-gon.

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

9.4.4 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 ]
```

9.4.5 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 ]
```

9.4.6 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 ] ]
```

9.4.7 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 10

Stratified Operations

10.1 Computational tools

I should say something more here.

10.1.1 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].

10.1.2 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.

10.1.3 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.

Example

```
gap> g:=AutomorphismGroup(Simplex(2));
<fp group of size 6 on the generators [ r0, r1 ]>
gap> AssignGeneratorVariables(g);
#I  Assigned the global variables [ r0, r1 ]
gap> SetReducedMultiplication(r0);
gap> s0:=[(1,2),[r0,r0,r1]];;s1:=[(2,3),[r0,r1,r1]];;
gap> ChunkGeneratedGroupElements([s0,s1],g);
[ [ (1,2), [ r0, r0, r1 ] ], [ (2,3), [ r0, r1, r1 ] ],
  [ (), [ <identity ...>, <identity ...>, <identity ...> ] ],
```

```
[ (1,3,2), [ <identity ...>, <identity ...>, r0*r1 ] ],
[ (1,2,3), [ r1*r0, <identity ...>, <identity ...> ] ], [ (1,3), [ r0, r0, r0 ] ],
[ (1,3), [ r1, r1, r1 ] ], [ (1,2,3), [ <identity ...>, r0*r1, r0*r1 ] ],
[ (1,3,2), [ r1*r0, r1*r0, <identity ...> ] ], [ (2,3), [ r0*r1*r0, r0, r0 ] ],
[ (1,2), [ r1, r1, r0*r1*r0 ] ], [ (), [ r0*r1, r0*r1, r0*r1 ] ],
[ (), [ r1*r0, r1*r0, r1*r0 ] ], [ (1,2), [ r0*r1*r0, r0*r1*r0, r0 ] ],
[ (2,3), [ r1, r0*r1*r0, r0*r1*r0 ] ], [ (1,3,2), [ r0*r1, r0*r1, r1*r0 ] ],
[ (1,2,3), [ r0*r1, r1*r0, r1*r0 ] ], [ (1,3), [ r0*r1*r0, r0*r1*r0, r0*r1*r0 ] ] ]
```

10.1.4 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
```

10.1.5 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 11

Maps On Surfaces

11.1 Bicontactual regular maps

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

11.1.1 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 ])
```

11.1.2 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);
ReflexibleManiplex([ 10, 2 ], "(r0 r1)^5 r2")
gap> Deltak(6);
ReflexibleManiplex([ 12, 2 ], "(r0 r1)^6 r2")
```

11.1.3 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);
ReflexibleManiplex([ 10, 5 ], "(r0 r1)^5 r0 = r2")
ReflexibleManiplex([ 12, 12 ], "(r0 r1)^6 r0 = r2")
```

11.1.4 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")
```

11.1.5 Bk2l (for IsInt,IsInt)

▷ `Bk2l(k, 2l)` (operation)

Returns: IsManiplex

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

Example
```
gap> Bk2l(4,10);
3-maniplex with 80 flags
```

11.1.6 Bk2lStar (for IsInt,IsInt)

▷ `Bk2lStar(k, 2l)` (operation)

Returns: IsManiplex

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

Example
```
gap> Bk2lStar(5,14);
3-maniplex with 140 flags
```

11.1.7 Bk2lRhoSigma (for IsInt,IsInt,IsInt,IsInt)

▷ `Bk2lRhoSigma(k, 2l, rho, sigma)` (operation)

Returns: IsPolytope

Given integers satisfying the constraints in [Wil85], this function will create the map $B(k,2l,\rho,\sigma)$.

Example
```
gap> Bk2lRhoSigma(4,16,3,0);
reflexible 3-maniplex
gap> IsSelfPetrial(last);
true
gap> m:=Bk2lRhoSigma(4,16,3,0);
reflexible 3-maniplex
```

```
gap> IsSelfPetrial(m);
true
gap> Opp(m)=Bk21RhoSigma(4,16,5,2);
true
gap> SchlafliSymbol(m);
[ 16, 4 ]
```

11.2 Operations on reflexible maps

11.2.1 Opp (for IsMapOnSurface)

▷ `Opp(map)` (operation)

Returns: IsManiplex

Forms the opposite map of the maniplex *map*.

Example
```
gap> Opp(Bk21Star(5,7));
Petrial(Dual(Petrial(3-maniplex with 140 flags)))
```

11.2.2 Hole (for IsMapOnSurface,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(Bk21Star(5,7),2);
3-maniplex with 140 flags
```

11.3 Map properties

`IsMapOnSurface` will test to see if you have rank 3 maniplex.

Example
```
gap> Filtered([HemiCube(3),Cube(4),Icosahedron()],IsMapOnSurface);
[ HemiCube(3), Icosahedron() ]
```

11.4 Operations on maps

11.4.1 Truncation (for IsMapOnSurface)

▷ `Truncation(map)` (operation)

Returns: trunc_map

Given a *map* on a surface, this function will return the truncation of *map*.

Example
```
gap> SchlafliSymbol(Truncation(Simplex(3)));
[ [ 3, 6 ], 3 ]
gap> TruncatedTetrahedron()=Truncation(Simplex(3));
```

```
true
gap> Truncation(CrossPolytope(3))=TruncatedOctahedron();
true
gap> Truncation(Cube(3))=TruncatedCube();
true
```

11.4.2 Snub (for IsMapOnSurface)

▷ Snub(*M*) (operation)

Returns: snub_map

Returns the snub of a given map; we require that the map have triangles as vertex figures.

Example
```
gap> Snub(Dodecahedron())=SnubDodecahedron();
true
gap> Snub(Cube(3))=SnubCube();
true
gap> Snub(Simplex(3))=Icosahedron();
true
gap> Snub(CrossPolytope(3))=SnubCube();
true
gap> Snub(Dual(Cube(3)))=Reflection(Snub(Reflection(Cube(3))));
true
```

11.4.3 Chamfer (for IsMapOnSurface)

▷ Chamfer(*M*) (operation)

Returns: chamfered_map

Returns the map obtained from the chamfering operation described in [dRF14]

Example
```
gap> s0 := (4,5)(6,7)(8,9);;
gap> s1 := (2,6)(3,4)(5,7);;
gap> s2 := (1,2)(4,8)(5,9);;
gap> poly := Group([s0,s1,s2]);;
gap> p:=ARP(poly);;
gap> SchlafliSymbol(p);
[ 6, 3 ]
gap> ch:=Chamfer(p);
3-maniplex with 432 flags
gap> SchlafliSymbol(ch);
[ 6, 3 ]
```

11.4.4 Subdivision1 (for IsMapOnSurface)

▷ Subdivision1(*M*) (operation)

Returns: Su1

Returns the One-dimensional subdivision of a map, which replaces each edge with two edges. For more information on the oriented version of this, see [BPW17].

Example
```
gap> m:=Subdivision1(Simplex(3));
3-maniplex with 48 flags
gap> SchlafliSymbol(m);
[ 6, [ 2, 3 ] ]
```

11.4.5 Subdivision2 (for IsMapOnSurface)

▷ `Subdivision2(M)` (operation)

Returns: Su2

Returns the two-dimensional subdivision of *M*.

Example
```
gap> SchlafliSymbol(Subdivision2(Cube(3)));
[ 3, [ 4, 6 ] ]
```

11.4.6 BarycentricSubdivision (for IsMapOnSurface)

▷ `BarycentricSubdivision(M)` (operation)

Returns: barycentric_subdivision

Gives the barycentric subdivision of *M*.

Example
```
gap> m:=BarycentricSubdivision(Cube(3));;
gap> SchlafliSymbol(m);NumberOfFacets(m);
[ 3, [ 4, 6, 8 ] ]
48
```

11.4.7 Leapfrog (for IsMapOnSurface)

▷ `Leapfrog(M)` (operation)

Returns: leapfrog

Gives the result of performing the leapfrog operation on a map on a surface

Example
```
gap> Leapfrog(Dodecahedron());
3-maniplex with 360 flags
gap> SchlafliSymbol(last);
[ [ 5, 6 ], 3 ]
```

11.4.8 CombinatorialMap (for IsMapOnSurface)

▷ `CombinatorialMap(M)` (operation)

Returns: combinatorial_map

Gives the result of combinatorial operation on a map; this is equivalent to taking the dual of the barycentric subdivision.

Example
```
gap> NumberOfEdges(Cube(3));
12
gap> NumberOfEdges(CombinatorialMap(Cube(3)));
72
```

11.4.9 Angle (for IsMapOnSurface)

▷ `Angle(M)` (operation)

Returns: angle_map

Returns the angle map of a map. This is equivalent to taking the dual of the medial.

Example
```
gap> NumberOfEdges(ToroidalMap44([3,0]));
18
gap> NumberOfEdges(Angle(ToroidalMap44([3,0])));
36
```

11.4.10 Gothic (for IsMapOnSurface)

▷ `Gothic(M)` (operation)

Returns: gothic

Returns the result of performing the gothic operation to a map. This is the same as taking the dual of the medial of the truncation of the map.

Example
```
gap> m:=AbstractRegularPolytope([ 3, 6 ], "(r0 r1 r2)^6");;
gap> NumberOfEdges(m); NumberOfEdges(Gothic(m));
27
162
```

11.5 Conway polyhedron operator notation

We include here operators from Wikipedia that are not included above.

- `MapJoin`: Creates quadrilateral faces by placing a node in each face, and then the set of edges are formed by the nodes corresponding to incident vertex-face pairs. This is another name for `Angle`.
- `Ambo`: This is another name for `Medial`.

Another excellent source for information on these types of operations is `https://antitile.readthedocs.io/en/latest/conway.html`. Additional functions are described below.

11.5.1 Reflection (for IsManiplex)

▷ `Reflection(M)` (operation)

Returns: reflection

Reverses the orientation of a maniplex.

Example
```
gap> m:=Cube(3);
Cube(3)
gap> Gyro(Dual(m))=Reflection(Gyro(Reflection(m)));
true
gap> Reflection(m)=EnantiomorphicForm(m);
true
gap> Reflection(Truncation(m))=Truncation(EnantiomorphicForm(m));
true
```

11.5.2 Kis (for IsMapOnSurface)

▷ Kis(*M*) (operation)

Returns: kis

Returns the Kis of the map, which erects a pyramid over each of the faces.

Example

```
gap> Kis(Cube(3));
3-maniplex with 144 flags
gap> SchlafliSymbol(last);
[ 3, [ 4, 6 ] ]
```

11.5.3 Needle (for IsMapOnSurface)

▷ Needle(*M*) (operation)

Returns: needle

Performs the needle operation to the map: edges connect adjacent face centers, and face centers to incident vertices.

Example

```
gap> SchlafliSymbol(Needle(Cube(3)));
[ 3, [ 3, 8 ] ]
```

11.5.4 Zip (for IsMapOnSurface)

▷ Zip(*M*) (operation)

Returns: zip

Returns the zip of the map.

Example

```
gap> Zip(Cube(3))=TruncatedOctahedron();
true
```

11.5.5 Ortho (for IsMapOnSurface)

▷ Ortho(*M*) (operation)

Returns: ortho

Returns the ortho of the map (this is the same as applying the join twice.).

Example

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

11.5.6 Expand (for IsMapOnSurface)

▷ Expand(*M*) (operation)

Returns: expand

Returns the expand of the map (this is the same as applying the ambo operation twice.).

Example

```
gap> Expand(Cube(3))=SmallRhombicuboctahedron();
true
```

11.5.7 Gyro (for IsMapOnSurface)

▷ `Gyro(M)` (operation)

Returns: gyro

Returns the gyro of the map.

Example
```
gap> Gyro(Dual(Cube(3)))=Gyro(Cube(3));
true
```

11.5.8 Meta (for IsMapOnSurface)

▷ `Meta(M)` (operation)

Returns: meta

Constructs the meta of the given map. (This is the same as applying first the join, and then the kis operation to the map).

Example
```
gap> Size(Cube(3))=NumberOfFacets(Meta(Cube(3)));
true
```

11.5.9 Bevel (for IsMapOnSurface)

▷ `Bevel(M)` (operation)

Returns: bevel

Constructs the bevel of a given map. (This is the same as truncating the ambo of a map.)

Example
```
gap> CombinatorialMap(Cube(3))=Bevel(Cube(3));
true
```

11.6 Extended operations

A number of these were introduced by George Hart.

11.6.1 Subdivide (for IsMapOnSurface)

▷ `Subdivide(M)` (operation)

Returns: u

Returns the subdivide (u) of `M`.

Example
```
gap> Chamfer(Dual(Cube(3)))=Dual(Subdivide(Cube(3)));
true
gap> SchlafliSymbol(Subdivide(Cube(3)));
[ [ 3, 4 ], [ 3, 6 ] ]
```

11.6.2 Propeller (for IsMapOnSurface)

▷ `Propeller(M)` (operation)

Returns: propeller

Constructs the propeller of the map.

Example
```
gap> Dual(Propeller(Cube(3)))=Propeller(Dual(Cube(3)));
true
gap> Dual(Propeller(Dual(Cube(3))))=Propeller(Cube(3));
true
```

11.6.3 Loft (for IsMapOnSurface)

▷ `Loft(M)` (operation)

Returns: loft

Constructs the loft of the map.

Example
```
gap> NumberOfFacets(Loft(Cube(3)));
30
gap> SchlafliSymbol(Loft(Cube(3)));
[ 4, [ 3, 6 ] ]
```

11.6.4 Quinto (for IsMapOnSurface)

▷ `Quinto(M)` (operation)

Returns: quinto

Constructs the quinto of the map.

Example
```
gap> SchlafliSymbol(Quinto(Cube(3)));
[ [ 4, 5 ], [ 3, 4 ] ]
```

11.6.5 JoinLace (for IsMapOnSurface)

▷ `JoinLace(M)` (operation)

Returns: join-lace

Constructs the join-lace of the map.

Example
```
gap> SchlafliSymbol(JoinLace(Cube(3)));
[ [ 3, 4 ], [ 4, 6 ] ]
```

11.6.6 Lace (for IsMapOnSurface)

▷ `Lace(M)` (operation)

Returns: lace

Constructs the lace of the map.

Example
```
gap> SchlafliSymbol(Lace(Cube(3)));
[ [ 3, 4 ], [ 4, 9 ] ]
```

11.6.7 Stake (for IsMapOnSurface)

▷ `Stake(M)` (operation)

Returns: stake

Constructs the stake of the map.

Example

```
gap> SchlafliSymbol(Stake(Cube(3)));
[ [ 3, 4 ], [ 3, 4, 9 ] ]
```

11.6.8 Whirl (for IsMapOnSurface)

▷ `Whirl(M)` (operation)

Returns: whirl

Constructs the whirl of the map.

Example

```
gap> SchlafliSymbol(Whirl(Cube(3)));
[ [ 4, 6 ], 3 ]            ^
gap> SchlafliSymbol(Whirl(Icosahedron()));
[ [ 3, 6 ], [ 3, 5 ] ]
```

11.6.9 Volute (for IsMapOnSurface)

▷ `Volute(M)` (operation)

Returns: volute

Constructs the volute of the map. This is equivalent to `Dual(Whirl(Dual(M)))`.

Example

```
gap> SchlafliSymbol(Volute(Cube(3)));
[ [ 3, 4 ], [ 3, 6 ] ]
gap> SchlafliSymbol(Volute(Dual(Cube(3))));
[ 3, [ 4, 6 ] ]
```

11.6.10 JoinKisKis (for IsMapOnSurface)

▷ `JoinKisKis(M)` (operation)

Returns: joinkiskis

Constructs the join-kis-kis of the map.

Example

```
gap> SchlafliSymbol(JoinKisKis(Cube(3)));
[ [ 3, 4 ], [ 3, 8, 9 ] ]
```

11.6.11 Cross (for IsMapOnSurface)

▷ `Cross(M)` (operation)

Returns: cross

Constructs the cross of the map.

Example

```
gap> SchlafliSymbol(Cross(Cube(3)));
[ [ 3, 4 ], [ 4, 6 ] ]
```

Chapter 12

Posets

12.1 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].

12.1.1 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 ] ], [
```

12.1.2 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.
```

12.1.3 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.
```

12.1.4 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.

12.1.5 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 ] ]
gap> ml:=["abc","abd","acd","bcd"];;
gap> p:=PosetFromAtomicList(ml);;
gap> List(Flags(p),x->List(x,AtomList));
[ [ [  ], "a", "ab", "abc", "abcd" ], [ [  ], "a", "ab", "abd", "abcd" ],
  [ [  ], "a", "ac", "abc", "abcd" ], [ [  ], "a", "ac", "acd", "abcd" ],
  [ [  ], "a", "ad", "abd", "abcd" ], [ [  ], "a", "ad", "acd", "abcd" ],
  [ [  ], "b", "ab", "abc", "abcd" ], [ [  ], "b", "ab", "abd", "abcd" ],
  [ [  ], "b", "bc", "abc", "abcd" ], [ [  ], "b", "bc", "bcd", "abcd" ],
  [ [  ], "b", "bd", "abd", "abcd" ], [ [  ], "b", "bd", "bcd", "abcd" ],
  [ [  ], "c", "ac", "abc", "abcd" ], [ [  ], "c", "ac", "acd", "abcd" ],
  [ [  ], "c", "bc", "abc", "abcd" ], [ [  ], "c", "bc", "bcd", "abcd" ],
  [ [  ], "c", "cd", "acd", "abcd" ], [ [  ], "c", "cd", "bcd", "abcd" ],
  [ [  ], "d", "ad", "abd", "abcd" ], [ [  ], "d", "ad", "acd", "abcd" ],
  [ [  ], "d", "bd", "abd", "abcd" ], [ [  ], "d", "bd", "bcd", "abcd" ],
  [ [  ], "d", "cd", "acd", "abcd" ], [ [  ], "d", "cd", "bcd", "abcd" ] ]
```

12.1.6 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) ]) ]
```

12.1.7 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 ], [ ] ]);
```

12.1.8 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.

12.1.9 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
```

12.1.10 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
```

12.1.11 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.
```

12.2 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].

12.2.1 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 ]
```

12.2.2 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.

12.2.3 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.

12.2.4 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
```

12.2.5 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).

12.2.6 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
```

12.2.7 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.

12.2.8 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
```

12.2.9 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.

12.2.10 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
```

12.2.11 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
```

12.2.12 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

12.2.13 IsFlagConnected (for IsPoset)

▷ `IsFlagConnected(poset)` (property)

Returns: `true` or `false`

Determines whether a poset is flag connected.

12.2.14 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.

12.2.15 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" ]
```

12.2.16 IsPrePolytope (for IsPoset)

▷ `IsPrePolytope(poset)` (property)

Returns: `true` or `false`

Determines whether a poset is an abstract pre-polytope.

12.2.17 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
```

12.3 Working with posets

12.3.1 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
```

12.3.2 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*.

12.3.3 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.

12.3.4 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.

12.3.5 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`.

12.3.6 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.

12.3.7 EqualChains (for IsList,IsList)

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

Determines whether two chains are equal.

12.3.8 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.

12.3.9 ConnectionGroup (for IsPoset)

▷ `ConnectionGroup(`*`poset`*`)` (attribute)

Returns: `IsPermGroup`

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

12.3.10 AutomorphismGroup (for IsPoset)

▷ `AutomorphismGroup(`*`poset`*`)` (attribute)

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

12.3.11 AutomorphismGroupOnElements (for IsPoset)

▷ `AutomorphismGroupOnElements(`*`poset`*`)` (attribute)

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

12.3.12 AutomorphismGroupOnChains (for IsPoset, IsCollection)

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

Returns: group

Returns the permutation group, representing the action of the automorphism group of *poset* on the chains of *poset* of type *I*.

Example
```
gap>
```

12.3.13 AutomorphismGroupOnIFaces (for IsPoset, IsInt)

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

Returns: group

Returns the permutation group, representing the action of the automorphism group of *poset* on the faces of *poset* of rank *I*.

12.3.14 AutomorphismGroupOnFacets (for IsPoset)

▷ `AutomorphismGroupOnFacets(poset)` (attribute)

Returns: group

Returns the permutation group, representing the action of the automorphism group of *poset* on the faces of *poset* of rank $d-1$.

12.3.15 AutomorphismGroupOnEdges (for IsPoset)

▷ `AutomorphismGroupOnEdges(poset)` (attribute)

Returns: group

Returns the permutation group, representing the action of the automorphism group of *poset* on the faces of *poset* of rank 1.

12.3.16 AutomorphismGroupOnVertices (for IsPoset)

▷ `AutomorphismGroupOnVertices(poset)` (attribute)

Returns: group

Returns the permutation group, representing the action of the automorphism group of *poset* on the faces of *poset* of rank 0.

12.3.17 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.

12.3.18 RankPosetElements (for IsPoset)

▷ `RankPosetElements(poset)` (operation)

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

12.3.19 FacesByRankOfPoset (for IsPoset)

▷ `FacesByRankOfPoset(poset)` (operation)

Returns: `list`

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

12.3.20 HasseDiagramOfPoset (for IsPoset)

▷ `HasseDiagramOfPoset(poset)` (operation)

Returns: directed graph

12.3.21 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
```

12.3.22 Max/min faces

▷ `MinFace(poset)` (operation)

▷ `MaxFace(arg)` (operation)

Returns: face

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

12.4 Element constructors

12.4.1 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 $b \leq a$.

12.4.2 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*.

12.4.3 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.

12.4.4 PosetElementFromIndex (for IsObject)

▷ `PosetElementFromIndex(obj)` (operation)

Returns: `IsFace`

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

12.4.5 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.

12.4.6 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.

12.4.7 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*.

12.4.8 FlagList (for IsPosetElement)

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

Returns: `list`

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

12.4.9 AtomList (for IsPosetElement)

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

Returns: `list`

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

12.5 Element operations

12.5.1 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`*.

12.5.2 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.

12.5.3 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.

12.5.4 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`.

12.5.5 Meet (for IsFace, IsFace, IsPoset)

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

Returns: meet

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

12.5.6 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.

12.6 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).

12.6.1 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
```

12.6.2 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
```

12.6.3 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
```

12.6.4 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
```

12.6.5 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 13

Graphs for Maniplexes

13.1 Graph families

13.1.1 HeawoodGraph

▷ `HeawoodGraph()` (operation)

Returns: `IsGraph`

Heawood Graph as described at https://www.distanceregular.org/graphs/heawood.html

13.1.2 PetersenGraph

▷ `PetersenGraph()` (operation)

Returns: `IsGraph`

Petersen Graph as described at https://www.gap-system.org/Manuals/pkg/grape/htm/CHAP002.htm

13.1.3 CirculantGraph (for IsInt,IsList)

▷ `CirculantGraph(n, L)` (operation)

Returns: `IsGraph`

Given an integer `n` and a list `L`, this returns the Circulant Graph with `n` vertices. For each `i` in the list `L` and each vertex `v`, there is an edge from `v` to `v+i` and `v-i` (mod `n`)

13.1.4 CompleteBipartiteGraph (for IsInt,IsInt)

▷ `CompleteBipartiteGraph(n, m)` (operation)

Returns: `IsGraph`

Given two integers `n, m`, this returns the Complete Bipartite Graph `K_{n,m}`.

13.2 Graph constructors for maniplexes

Note that this functionality depends on the functionality of the GRAPE package.

13.2.1 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 ] )
```

13.2.2 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 ] )
```

13.2.3 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));
```

13.2.4 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));
```

13.2.5 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);;
```

13.2.6 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());;
```

13.2.7 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
```

13.2.8 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 ] )
```

13.2.9 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
```

13.2.10 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);
```

13.2.11 EdgeLabeledGraphFromLabeledEdges (for IsList)

▷ `EdgeLabeledGraphFromLabeledEdges(list)` (operation)

Returns: `IsEdgeLabeledGraph`.

Given a list of labeled edges this represents the edge labeled (multi)-graph with those parameters. Semi-edges are represented by a singleton in the edge list.

Example

```
L:=[[[1],0],[[2],0],  [ [1,2],1]];;
X2:=EdgeLabeledGraphFromLabeledEdges(L);
```

13.2.12 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 premaniplex, then the connection group is calculated.

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

Example
```
gap> C:=ConnectionGroup(Simplex(3));;
gap> gamma:=FlagGraph(C);
Edge labeled graph with 24 vertices, and edge labels [ 0, 1, 2 ]
gap> STG3(4,1);;
gap> FlagGraph(last);
Edge labeled graph with 3 vertices, and edge labels [ 0, 1, 2, 3 ]
```

13.2.13 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)));
```

13.2.14 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);
```

13.2.15 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.

13.2.16 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]);
```

13.2.17 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);
```

13.2.18 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.

13.2.19 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.

13.2.20 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.

13.2.21 SemiEdges (for IsEdgeLabeledGraph)

▷ `SemiEdges(EdgeLabeledGraph)` (attribute)

Returns: `IsList`.

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

13.2.22 LabeledSemiEdges (for IsEdgeLabeledGraph)

▷ `LabeledSemiEdges(EdgeLabeledGraph)` (attribute)

Returns: `IsList, IsList`.

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

13.2.23 LabeledDarts (for IsEdgeLabeledGraph)

▷ `LabeledDarts(EdgeLabeledGraph)` (attribute)

Returns: `IsList`.

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

13.2.24 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 ]
```

13.2.25 ViewGraph (for IsObject, IsString)

▷ `ViewGraph(G, software_name)` (operation)

Returns: `IsString`.

Given a Graph or EdgeLabeledGraph `G`, outputs code to view the graph in other software. Currently Mathematica and Sage are supported. If the software is not specified it will return the code for Mathematica.

13.2.26 ConnectionGroup (for IsEdgeLabeledGraph)

▷ `ConnectionGroup(F)` (attribute)

Returns: `IsPermGroup`

Constructs the connection group from an edge labeled graph. Loops, semi-edges, and non-edges give fixed points. Graph is assumed to be coming from a maniplex. Some weird things could happen if it is not

13.2.27 FlagGraph (for IsPremaniplex)

▷ `FlagGraph(M)` (operation)

Returns: edgelabeledgraph

Returns the flag graph of a premaniplex `M`.

Example

```
gap> STG3(4,1);;
 gap> FlagGraph(last);
Edge labeled graph with 3 vertices, and edge labels [ 0, 1, 2, 3 ]
```

13.2.28 LabeledDarts (for IsPremaniplex)

▷ `LabeledDarts(M)` (attribute)

Returns: list

Given a Premaniplex *M*, returns the list of labeled darts from its flag graph.

Example

```
gap> P:=STG2(5,[2,4]);;
gap> LabeledDarts(P);
[ [ [ 1, 2 ], 0 ], [ [ 2, 1 ], 0 ], [ [ 1, 2 ], 1 ], [ [ 2, 1 ], 1 ], [ [ 1 ], 2 ], [ [ 1, 2 ], 3
```

Chapter 14

Voltage Graphs and Operations

14.1 Voltage Operator

14.1.1 VoltageOperator (for IsList, IsString,IsEdgeLabeledGraph)

▷ VoltageOperator(*etain*, *etaout*, *Xa*) (operation)

Returns: IsManiplex

Returns the output of the voltage operator acting on Xa. Xa is a n-premaniplex as an edge labeled graph, Y is a m-premaniplex. eta is a voltage assignment on the darts of Y. etain is a list of all darts of Y. etaout is a string giving words in the universal sggi of rank n, and the order of the words corresponds to the order of the darts in etain. If Xa is given as a maniplex, the operation is done to its flag graph.

14.1.2 VoltageOperator (for IsList, IsString,IsManiplex)

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

Example

```
The Petrial and the dual can be built using voltage operations
Similarly for rank 3 other operations can be built this way.
See VOLTAGE OPERATIONS ON MANIPLEXES by HUBARD, MOCHÁN, MONTERO
gap> etain1:=[[[1],0],[[1],1],[[1],2],[[1],3]];;
gap> etain2:=[[[1],0],[[2],0],[[1],1],[[2],1],[[1,2],2]];;
gap> etain3:=[[[1],0],[[2],0],[[3],0],[[1],1],[[3],2],[[1,2],2],[[2,3],1]];;
gap> etaoutPetrial:="r0, r1 r3, r2, r3";;
gap> etaoutDual:="r3, r2, r1, r0";;
gap> etaoutMedial:="r1, r1, r0, r2, Id";;
gap> etaoutLeapfrog:="r1,r1,r2,r0,r0, , ";;
gap> etaoutTruncation:="r1, r1, r0, r2, r2,Id, Id";;
gap> Petrial(Cube(4)) =VoltageOperator(etain1, etaoutPetrial, Cube(4));
true
gap> Dual(Cube(4)) = VoltageOperator(etain1, etaoutDual, Cube(4));
true
gap> Medial(Dodecahedron()) = VoltageOperator(etain2, etaoutMedial, Dodecahedron());
true
gap> Leapfrog(Simplex(3)) =  VoltageOperator(etain3, etaoutLeapfrog, Simplex(3));
true
```

```
gap> Truncation(Prism(7)) = VoltageOperator(etain3, etaoutTruncation, Prism(7));
true
```

14.1.3 AdmissiblePerms (for IsInt, IsList)

▷ AdmissiblePerms(*n*, *I*) (operation)

Returns: IsList

Returns a list of the admissible sequences that correspond to the flag orbits for a Wythoffian of a rank `n` maniplex. The vertex in the fundamental region is moved by ri for i in `I`.

Example

```
There will be three flag orbits in the truncation of a rank 3 maniplex, where truncation is a Wyt
gap> AdmissiblePerms(3,[0,1]);
 [ [ 0, 1, 2 ], [ 1, 0, 2 ], [ 1, 2, 0 ] ]
```

14.1.4 WythoffSTG (for IsInt, IsList)

▷ WythoffSTG(*n*, *I*) (operation)

Returns: IsList

Returns the symmetry type graph for a Wythoffian of rank `n` defined by a list of indices `I`. See, for instance, VOLTAGE OPERATIONS ON MANIPLEXES.

Example

```
Symmetry type graph of a medial operation
gap> W:=WythoffSTG(3,[1]);
Edge labeled graph with 2 vertices, and edge labels [ 0, 1, 2 ]
gap> LabeledEdges(W);
[ [ [ 1 ], 0 ], [ [ 1 ], 1 ], [ [ 1, 2 ], 2 ], [ [ 2 ], 0 ], [ [ 2 ], 1 ] ]
```

14.1.5 WythoffLabeledEdges (for IsInt, IsList)

▷ WythoffLabeledEdges(*n*, *I*) (operation)

Returns: IsList

Returns the labeled edges of a possible symmetry type graph for a Wythoffian of rank `n` defined by a list of indices `I`. The actual graph is not returned, as we require edge labeled graphs to have integer vertices in order to calculate their connection groups.

Example

```
Labeled Edges of the Symmetry type graph of a medial operation
gap> WythoffLabeledEdges(3,[1]);
[ [ [ [ 1, 0, 2 ] ], 0 ], [ [ [ 1, 0, 2 ] ], 1 ], [ [ [ 1, 2, 0 ] ], 0 ], [ [ [ 1, 2, 0 ] ], 1 ],
```

14.1.6 Wythoffian (for IsList, IsManiplex)

▷ Wythoffian(*I*, *M*) (operation)

Returns: IsList

Returns the Wythoffian of the maniplex `M` with set of ringed nodes `I`. For example, if I = [1] then this is the rectification of `M` and if I = [0, 1] then this is truncation of `M`. Behind the scenes, this is accomplished using VoltageOperator.

```
Example
gap> W:=Wythoffian([0,1],Dodecahedron());
3-maniplex with 360 flags
gap> W=Truncation(Dodecahedron());
true
gap> M := Wythoffian([1], Simplex(4));;
gap> Fvector(M);
[10, 30, 30, 10]
gap> VertexFigure(M) = Prism(3);
true
gap> Wythoffian([3], M) = Dual(M);
true
```

14.1.7 VoltageGraph (for IsGroup,IsList,IsList)

▷ `VoltageGraph(G, L, V)` (operation)

Returns: IsVoltageGraph

Given an IsGroup *G*, an IsList *L*, and an IsList *V*, `VoltageGraph(G,L,V)` will construct the voltage graph with voltages from G, labeled darts from L, and voltages from V.

```
Example
gap> G:=ConnectionGroup(Cube(3));;
gap> L:=[ [[1],0], [[1],1], [[1,2],2], [[2],0], [[2],1]];;
gap> V:=[G.2, G.1, Identity(G), G.2, G.1];;
gap> VG:=VoltageGraph(G,L,V);
Voltage Graph with voltages from Group( [ (1,20)(2,13)(3,10)(4,45)(5,35)(6,7)(8,41)(9,28)(11,38)
(12,24)(14,43)(15,34)(16,33)(17,19)(18,31)(21,39)(22,27)(23,26)(25,36)(29,32)(30,48)(37,47)
(40,46)(42,44), (1,11)(2,32)(3,14)(4,25)(5,26)(6,27)(7,8)(9,43)(10,44)(12,29)(13,30)(15,39)(16,40
(17,41)(18,21)(19,22)(20,23)(24,48)(28,42)(31,47)(33,36)(34,37)(35,38)(45,46), (1,3)(2,7)
(4,11)(5,12)(6,13)(8,18)(9,19)(10,20)(14,25)(15,26)(16,27)(17,28)(21,32)(22,33)(23,34)(24,35)
(29,39)(30,40)(31,41)(36,43)(37,44)(38,45)(42,47)(46,48) ] )
```

14.1.8 VoltageGraph (for IsGroup,IsPremaniplex,IsList)

▷ `VoltageGraph(G, P, V)` (operation)

Returns: IsVoltageGraph

Given an IsGroup *G*, an IsPremaniplex *P*, and an IsList *V*, `VoltageGraph(G,P,V)` will construct the voltage graph with voltages from G, labeled darts from the premaniplex P, and voltages from V.

```
Example
gap> G:=ConnectionGroup(Cube(3));;
gap> P:=STG2(3,[0,1]);
Premaniplex of rank 3 with 2 flags
gap> L:=LabeledDarts(P);
[ [ [ 1 ], 0 ], [ [ 1 ], 1 ], [ [ 1, 2 ], 2 ], [ [ 2, 1 ], 2 ], [ [ 2 ], 0 ], [ [ 2 ], 1 ] ]
gap> V:=[G.2, G.1, Identity(G), Identity(G), G.2, G.1];;
gap> VG:=VoltageGraph(G,P,V);
Voltage Graph with voltages from Group( [ (1,20)(2,13)(3,10)(4,45)(5,35)(6,7)(8,41)(9,28)(11,38)(
(15,34)(16,33)(17,19)(18,31)(21,39)(22,27)(23,26)(25,36)(29,32)(30,48)(37,47)(40,46)(42,44),
(1,11)(2,32)(3,14)(4,25)(5,26)(6,27)(7,8)(9,43)(10,44)(12,29)(13,30)(15,39)(16,40)(17,41)(18,21)(
(24,48)(28,42)(31,47)(33,36)(34,37)(35,38)(45,46),
```

```
(1,3)(2,7)(4,11)(5,12)(6,13)(8,18)(9,19)(10,20)(14,25)(15,26)(16,27)(17,28)(21,32)(22,33)(23,34)(
(30,40)(31,41)(36,43)(37,44)(38,45)(42,47)(46,48) ] )
```

14.1.9 VoltageGraph (for IsGroup,IsPremaniplex)

▷ `VoltageGraph(G, P)` (operation)

Returns: IsVoltageGraph

Given an IsGroup `G`, and an IsPremaniplex `P`, `VoltageGraph(G,P)` will construct the voltage graph with voltages from G, labeled darts from the premaniplex P, and trivial voltages.

14.1.10 ChangeVoltage (for IsVoltageGraph,IsList, IsObject)

▷ `ChangeVoltage(VG, ld, g)` (operation)

Given an IsVoltageGraph `VG`, an IsList `ld`, and an IsObject `g`, `ChangeVoltage(VG,ld,g)` will change the voltage for the one labeled dart ld to the group element g.

14.1.11 ChangeVoltage (for IsVoltageGraph,IsInt,IsInt, IsObject)

▷ `ChangeVoltage(VG, lab, startvert, g)` (operation)

Given an IsVoltageGraph `VG`, an IsInt `lab`, an IsInt `startvert`, and an IsObject `g`, `ChangeVoltage(VG,lab, startvert,g)` will change the voltage for the one labeled dart of label lab and start vertex startvert to the group element g.

14.1.12 DerivedGraph (for IsVoltageGraph)

▷ `DerivedGraph(VG)` (attribute)

Returns: IsVoltageGraph

Given an IsVoltageGraph `VG`, a `DerivedGraph(VG)` will construct the derived graph of the voltage graph VG.

14.1.13 VoltageOperator (for IsVoltageGraph, IsManiplex)

▷ `VoltageOperator(VG, M)` (operation)

Given an IsVoltageGraph `VG`, and an IsManiplex `M`, `VoltageOperator(VG,M)` will return the voltage operator VG acting on M.

Example

```
gap> M:=Dodecahedron();;
gap> S:=STG2(3,[0,1]);
Premaniplex of rank 3 with 2 flags
gap> C:=ConnectionGroup(M);;
gap> V:=VoltageGraph(C,S);;
gap> ChangeVoltage(V,0,1,C.2);;
gap> ChangeVoltage(V,0,2,C.2);;
gap> ChangeVoltage(V,1,1,C.1);;
gap> ChangeVoltage(V,1,2,C.3);;
```

```
gap> Medial(M) = VoltageOperator(V,M);
true
```

14.1.14 VoltageOperator (for IsVoltageGraph,IsEdgeLabeledGraph)

▷ `VoltageOperator(VG, ELG)` (operation)

Given an IsVoltageGraph *VG*, and an IsEdgeLabeledGraph *ELM*, `VoltageOperator(VG,M)` will return the voltage operator VG acting on ELM.

Chapter 15

Databases

We are indebted to those who have made their data on polytopes and maps freely available. Data on small regular polytopes is from Marston Conder:

https://www.math.auckland.ac.nz/~conder/RegularPolytopesWithUpTo4000Flags-ByOrder.txt

Data on small reflexible maniplexes was produced for RAMP by Mark Mixer.

Data on small chiral polytopes is from Marston Conder:

https://www.math.auckland.ac.nz/~conder/ChiralPolytopesWithUpTo4000Flags-ByOrder.txt

Data on small chiral maps is from Primoz Potocnik:

https://users.fmf.uni-lj.si/potocnik/work.htm

Data on small 2-orbit polyhedra in class 2_0 (available in Rank3AG_2_0.txt in the data folder) was produced for RAMP by Mark Mixer.

15.1 Regular polyhedra

15.1.1 WriteManiplexesToFile

▷ `WriteManiplexesToFile(`*`maniplexes, filename, attributeNames`*`)` (function)

Writes the data in *`maniplexes`* to the designated file, including the defining information and the values of the attributes in *`attributeNames`*. This calls `DatabaseString` on each maniplex in *`maniplexes`* to get the file representation.

15.1.2 ManiplexesFromFile

▷ `ManiplexesFromFile(`*`filename`*`)` (function)

Returns: IsList

Reads the maniplexes from *`filename`* in the data directory of RAMP and returns them as a list. Note that for performance reasons, some safety checks are disabled for data read from a file. For example, `AbstractRegularPolytope` usually checks its input to make sure that it defines a polytope, but `ManiplexesFromFile` just assumes that any maniplex defined using `AbstractRegularPolytope` really is a polytope.

15.1.3 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 ]) ]
```

15.1.4 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 ]) ]
```

15.1.5 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") ]
```

15.1.6 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") ]
```

15.1.7 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 ") ]
```

15.1.8 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
```

15.1.9 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") ]
```

15.1.10 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 ]) ]
```

15.1.11 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") ]
```

15.1.12 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*
```

15.1.13 SmallReflexible3Maniplexes

▷ `SmallReflexible3Maniplexes(`*`sizerange`*`)` (function)

Returns: IsList

Gives all regular 3-maniplexes 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.

15.1.14 SmallChiral3Maniplexes

▷ `SmallChiral3Maniplexes(`*`sizerange`*`)` (function)

Returns: IsList

Gives all chiral 3-maniplexes with sizes in *`sizerange`* flags. Currently supports a `maxsize` of 12000 or less.

15.1.15 SmallReflexibleManiplexes

▷ `SmallReflexibleManiplexes(n, sizerange[, filt1, filt2, ...])` (function)

Returns: IsList

First finds a list of all reflexible maniplexes of rank *n* where the number of flags is in *sizerange*. Then applies the given filters and returns the result. Each filter is either a function-value pair or a boolean function. In the first case, we keep only those maniplexes such that applying the given function returns the given value. In the second case, we keep only those maniplexes such that the given boolean function returns `true`.

Example

```
gap> L := SmallReflexibleManiplexes(3, [100..200], IsPolytopal, [NumberOfVertices, 6]);;
gap> Size(L);
14
gap> ForAll(L, IsPolytopal);
true
gap> List(L, NumberOfVertices);
[ 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6 ]
```

15.1.16 SmallTwoOrbitPolyhedra

▷ `SmallTwoOrbitPolyhedra(I, sizerange)` (function)

Returns: IsList

Gives all two-orbit polyhedra in class 2_I with sizes in *sizerange* flags. Currently supports a `maxsize` of 1000 or less.

Example

```
gap> L := SmallTwoOrbitPolyhedra([0], 100);
[ TwoOrbit3ManiplexClass2_0([ 10, 4 ], "  r0*a21*a101*a21^-1, r0*a21^-1*a101*r0*a101*a21 "),
  TwoOrbit3ManiplexClass2_0([ 14, 3 ], "  r0*a21*a101*a21^-1, r0*a101*a21*(a101*r0)^2*a21^-1 ") ]
```

15.2 System internal representations

15.2.1 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. This works for any maniplex such that String(*M*) contains defining information for *M* - otherwise the output may not be so useful.

Example

```
gap> DatabaseString(Cube(3));
"Cube(3)#6#48"
gap> M := ReflexibleManiplex(Group((1,2),(2,3),(3,4)));;
gap> DatabaseString(M);
"<object>#4#24"
```

15.2.2 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. In particular, ManiplexFromDatabaseString(DatabaseString(M)) is isomorphic to M if DatabaseString(M) contains defining information for M.

Example
```
gap> ManiplexFromDatabaseString("Cube(3)#6#48") = Cube(3);
true
gap> M := ReflexibleManiplex(Group((1,2),(2,3),(3,4)));;
gap> ManiplexFromDatabaseString(DatabaseString(M));
Syntax error: expression expected in stream:1
_EVALSTRINGTMP:=<object>;
```

15.2.3 InterpolatedString (for IsString)

▷ `InterpolatedString(str)` (operation)

Returns: IsString

Given a string, replaces each instance of "$variable" with String(EvalString(variable)). Any character which cannot be used in a variable name (such as spaces, commas, etc.) marks the end of the variable name.

Note that, due to limitations with EvalString, only global variables can be interpolated this way.

Example
```
gap> n := 5;;
gap> InterpolatedString("2 + 3 = $n");
"2 + 3 = 5"
gap> InterpolatedString("2 + 3 = $n, right?");
"2 + 3 = 5, right?"
gap> nn := 17;;
gap> InterpolatedString("$n and $nn are different");
"5 and 17 are different"
```

Chapter 16

Utility Functions

16.1 System

16.1.1 InfoRamp

▷ `InfoRamp` (info class)

The InfoClass for the Ramp package. This is sort of an "information channel" that functions can send updates to, and by default, users of Ramp will see these messages. To add such a message to a function that you are writing for Ramp, use `Info(InfoRamp, 1, "This is a message!");`. For example, if you have a function `f` with this line, then the user will see this:

Example
```
gap> f(3);;
#I This is a message!
```

To turn off messages from this class, use `SetInfoLevel(InfoRamp, 0)`.

16.2 Polytopes

16.2.1 AbstractPolytope

▷ `AbstractPolytope(args)` (function)

Calls `Maniplex(args)` and verifies whether the output is polytopal. If not, this throws an error. Use `AbstractPolytopeNC` to assume that the output is polytopal and mark it as such.

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

16.2.2 AbstractRegularPolytope

▷ `AbstractRegularPolytope(args)` (function)

Calls `ReflexibleManiplex(args)` and verifies whether the output is polytopal. If not, this throws an error. Use `AbstractRegularPolytopeNC` to assume that the output is polytopal and mark it as such. Also available as `ARP(args)` and `ARPNC(args)`.

```
Example
gap> Pgon(5)=AbstractRegularPolytope(Group([(2,3)(4,5),(1,2)(3,4)]));
true
```

16.2.3 AbstractRotaryPolytope

▷ AbstractRotaryPolytope(*args*) (function)

Calls `RotaryManiplex(args)` and verifies whether the output is polytopal. If not, this throws an error. Use `AbstractRotaryPolytopeNC` to assume that the output is polytopal and mark it as such.

```
Example
gap> M := AbstractRotaryPolytope(Group((1,2)(3,4), (1,4)(2,3)));
regular 3-polytope of type [ 2, 2 ] with 8 flags
gap> M := AbstractRotaryPolytope(Group((1,2,3,4), (1,2)));
Error, The given group is not a String Rotation Group...
```

16.3 Permutations

16.3.1 TranslatePerm

▷ TranslatePerm(*perm, k*) (function)

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

```
Example
gap> TranslatePerm((1,2,3,4),5);
(6,7,8,9)
```

16.3.2 MultPerm

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

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

```
Example
gap> MultPerm((1,2,3)(4,5,6),3,7);
(1,2,3)(4,5,6)(8,9,10)(11,12,13)(15,16,17)(18,19,20)
gap> MultPerm((1,2,3,4),2,4);
(1,2,3,4)(5,6,7,8)
```

16.3.3 InvolutionListList

▷ InvolutionListList(*list1, list2*) (function)

Returns: involution

Construction the involution (when possible) with entries `(list1[i],list2[i])`.

```
Example
gap> InvolutionListList([3,4,5],[6,7,8]);
(3,6)(4,7)(5,8)
```

16.3.4 PermFromRange (for IsPerm, IsPerm, IsPerm)

▷ PermFromRange(*perm1*[, *perm2*], *perm3*) (operation)

Returns: Permutation

Given three permutations, where *perm2* and *perm3* are translations of *perm1*, forms the permutation that we would most likely denote by perm1 * perm2 * ... * perm3. Namely, if *perm2* is a translation of *perm1* by k, then we successively translate by k until we get *perm3*, and then we multiply those permutations together.

When only two permutations are given, then *perm2* is the smallest translation of perm1 such that `SmallestMovedPoint(perm2) > LargestMovedPoint(perm1)`.

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)
gap> PermFromRange((1,2), (101,102), (601,602));
(1,2)(101,102)(201,202)(301,302)(401,402)(501,502)(601,602)
```

16.4 Words on relations

16.4.1 ParseGgiRels

▷ ParseGgiRels(*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 := ParseGgiRels(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).

16.4.2 ParseRotGpRels

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

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

Example
```
gap> g := UniversalRotationGroup([4,4]);
<fp group of size infinity on the generators [ s1, s2 ]>
gap> rels := "(s1 s2^-1)^6";;
gap> newrels := ParseRotGpRels(rels, g);
[ (s1*s2^-1)^6 ]
gap> g2 := FactorGroupFpGroupByRels(g, newrels);
<fp group on the generators [ s1, s2 ]>
gap> M := RotaryManiplex(g2);
3-maniplex with 288 flags
gap> M = ToroidalMap44([6,0]);
true
```

16.4.3 StandardizeSggi

▷ StandardizeSggi(*g*) (function)

Returns: IsSggi

Takes an sggi, and returns an isomorphic sggi that is a quotient of the universal sggi of the appropriate rank.

Example
```
gap> f := FreeGroup("x","y","z");
<free group on the generators [ x, y, z ]>
gap> AssignGeneratorVariables(f);
#I  Assigned the global variables [ x, y, z ]
gap> g := f / [x^2, y^2, z^2, (x*z)^2, (x*y)^4, (y*z)^4, (x*y*z)^6];
<fp group on the generators [ x, y, z ]>
gap> IsSggi(g);
true
gap> g2 := StandardizeSggi(g);
<fp group on the generators [ r0, r1, r2 ]>
gap> ReflexibleManiplex(g) = ReflexibleManiplex(g2);
true
```

16.4.4 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];
```

16.4.5 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.

16.4.6 MarkAsPolytopal (for IsManiplex)

▷ `MarkAsPolytopal(M)` (operation)

Sets `IsPolytopal(M)` as true, and if necessary, changes `String(M)` to reflect this.

16.4.7 ReallyNaturalHomomorphismByNormalSubgroup (for IsGroup,IsGroup)

▷ `ReallyNaturalHomomorphismByNormalSubgroup(G, N)` (operation)

Returns: quotient group with generators appropriately mapped

Image(NaturalHomomorphismByNormalSubgroup(G,N)) tries to make the quotient efficient in terms of the number of generators, which is disastrous for studying Sggis. This fixes that.

16.4.8 ActionByGenerators

▷ `ActionByGenerators(G, S, act)` (function)

Returns a permutation group that represents the action of `G` on `S` as given by the action `act`. Furthermore, the generators of this permutation group are the images of the generators of `G`.

Example
```
gap> g := Group([ (1,2)(3,4)(5,6)(7,8), (2,3)(6,7), (3,5)(4,6) ]);;
gap> ActionByGenerators(g, [[1,8],[2,7],[3,6],[4,5]], OnSets);
Group([ (1,2)(3,4), (2,3), (3,4) ])
```

16.4.9 ActionOnBlocks

▷ `ActionOnBlocks(G, S, B)` (function)

Given a group `G` acting on a set `S` and an initial block `B`, returns the action of `G` on the block system induced by `B`. This is equivalent to `ActionByGenerators(G, Blocks(G, S, B), OnSets)`.

Example
```
gap> g := Group([ (1,2)(3,4)(5,6)(7,8), (2,3)(6,7), (3,5)(4,6) ]);;
gap> ActionOnBlocks(g, [1..8], [1,8]);
Group([ (1,2)(3,4), (2,3), (3,4) ])
```

16.4.10 VerifyProperties

▷ `VerifyProperties(M)` (function)

Returns: Boolean

Given a maniplex *M*, recalculates all of the stored properties (boolean attributes) and some of the stored numeric attributes of *M*. Returns true if the recalculated values agree with the stored values. Otherwise, outputs a list of which values had discrepancies and then returns false.

Note that the way that we recalculate the properties is to build a new maniplex from ConnectionGroup(*M*). So if this connection group is incorrect, then this method will not work as intended.

Example
```
gap> M := Maniplex(ConnectionGroup(Cube(3)));;
gap> SetNumberOfFlagOrbits(M, 3);
gap> VerifyProperties(M);
Value mismatch in NumberOfFlagOrbits: stored value is 3 and real value is 1
false
```

Chapter 17

Synonyms for Commands

Here we list, in alphabetical order, synonyms for common commands.

- `Ambo` for `Medial` (**RAMP: Medial for IsManiplex**)
- `AreIncidentFaces` for `AreIncidentElements` (**RAMP: AreIncidentElements for IsObject,IsObject**)
- `ARP` for `AbstractRegularPolytope` (**RAMP: AbstractRegularPolytope**)
- `Faces` for `ElementsList` (**RAMP: ElementsList for IsPoset**)
- `FacesList` for `ElementsList` (**RAMP: ElementsList for IsPoset**)
- `Flags` for `MaximalChains` (**RAMP: MaximalChains for IsPoset**)
- `FlagsList` for `MaximalChains` (**RAMP: MaximalChains for IsPoset**)
- `IsDiamondCondition` for `IsP4` (**RAMP: IsP4 for IsPoset**)
- `IsStronglyFlagConnected` for `IsP3` (**RAMP: IsP3 for IsPoset**)
- `MapJoin` for `Angle` (**RAMP: Angle for IsMapOnSurface**)
- `MonodromyGroup` for `ConnectionGroup` (**RAMP: ConnectionGroup for IsPremaniplex**)
- `NumberOfFlags` for `Size` (**RAMP: Size for IsPremaniplex**)
- `PetrieDual` for `Petrial` (**RAMP: Petrial for IsManiplex**)
- `RankPosetFaces` for `RankPosetElements` (**RAMP: RankPosetElements for IsPoset**)
- `RefMan` for `ReflexibleManiplex` (**RAMP: ReflexibleManiplex**)

References

[BPW17] Leah Wrenn Berman, Tomaž Pisanski, and Gordon Ian Williams. Operations on oriented maps. *Symmetry*, 9(11):274, 1–14, November 2017. 80

[CDRFHT15] Gabe Cunningham, María Del Río-Francos, Isabel Hubard, and Micael Toledo. Symmetry type graphs of polytopes and maniplexes. *Ann. Comb.*, 19(2):243–268, 2015. 30, 31

[CM17] Gabe Cunningham and Mark Mixer. Internal and external duality in abstract polytopes. *Contrib. Discrete Math.*, 12(2):187–214, 2017. 29, 30, 66, 67

[CPW22] Gabe Cunningham, Daniel Pellicer, and Gordon Ian Williams. Stratified operations on maniplexes. *Algebr. Comb.*, 2022. 75

[Cun21] Gabe Cunningham. Flat extensions of abstract polytopes. *Art Discrete Appl. Math.*, 4(3):Paper No. 3.06, 14, 2021. 65

[dRF14] María del Río Francos. Chamfering operation on k-orbit maps. *Ars Math. Contemp.*, 7(2):507–524, 2014. 80

[GH18] Ian Gleason and Isabel Hubard. Products of abstract polytopes. *Journal of Combinatorial Theory, Series A*, 157:287–320, jul 2018. 102

[GVH18] Jorge Garza-Vargas and Isabel Hubard. Polytopality of maniplexes. *Discrete Math.*, 341(7):2068–2079, 2018. 23

[HW10] Michael I. Hartley and Gordon I. Williams. Representing the sporadic archimedean polyhedra as abstract polytopes. *Discrete Mathematics*, 310(12):1835–1844, jun 2010. 34

[MPW12] Barry Monson, Daniel Pellicer, and Gordon Williams. The tomotope. *Ars Mathematica Contemporanea*, 5(2):355–370, jun 2012. 32

[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. 87

[MS02] Peter McMullen and Egon Schulte. *Abstract Regular Polytopes*. Cambridge University Press, dec 2002. 87, 91

[Pel18] Daniel Pellicer. Cleaved abstract polytopes. *Combinatorica*, 38(3):709–737, mar 2018. 91

[PW18] Daniel Pellicer and Gordon Ian Williams. Pyramids over regular 3-tori. *SIAM Journal on Discrete Mathematics*, 32(1):249–265, jan 2018. 74

[Wil85] Stephen Wilson. Bicontactual regular maps. *Pacific Journal of Mathematics*, 120(2):437–451, dec 1985. 77, 78

[Wil12] Steve Wilson. Maniplexes: Part 1: Maps, polytopes, symmetry and operators. *Symmetry*, 4(2):265–275, apr 2012. 87

Index